Center for Reliable Computing

TECHNICAL REPORT

ED^{4}I:

Error Detection by Diverse Data and Duplicated Instructions

Nahmsuk Oh, Subahshish Mitra and Edward J. McCluskey

CRC-TR 00 - 8

November, 2000

Center for Reliable Computing
Gates Building 2A, Room 236
Computer Systems Laboratory
Dept. of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-9020

ABSTRACT

Errors in computer systems can cause abnormal behavior and degrade data integrity and system availability. Fault avoidance techniques such as radiation hardening and shielding have been the major approaches to protecting the system from transient errors, but these techniques are expensive. Recently, unhardened *Commercial Off-The-Shelf* (COTS) components have been investigated for a low cost alternative to fault avoidance techniques, and *Software Implemented Hardware Fault Tolerance* (SIHFT) has been proposed to increase the data integrity and availability of COTS systems.

ED^{4}I is a SIHFT technique that detects both permanent and temporary errors by executing two "different" programs (with the same functionality) and comparing their outputs. ED^{4}I maps each number, x, in the original program into a new number x', and then transforms the program so that it operates on the new numbers so that the results can be mapped backwards for comparison with the results of the original program. The mapping in the transformation of ED^{4}I is $x' = k \cdot x$ for integer numbers, where k determines the fault detection probability and data integrity of the system.

We have developed a transformation algorithm for ED^{4}I and demonstrated how to choose an optimal value of k for the transformation.

This paper shows that for integer programs, the transformation with k = -2 was the most desirable choice in six out of seven benchmark programs simulated. It maximizes fault detection probability under the condition that data integrity is the highest. For programs that use floating point numbers, we need two transformed programs: the first one with $k_1 = -\frac{3}{2} \times 2^{10101010_2}$ and the second one with $k_2 = -\frac{3}{2} \times 2^{01010101_2}$. We demonstrate these results by both theoretical and simulation analysis.

FUNDING: This work was supported in part by the Ballistic Missile Defense Organization, Innovative Science and Technology (BMDO/IST) Directorate and administered through the Department of the Navy, Office of Naval Research under Grant Nos. N00014-92-J-1782 and N00014-95-1-1047

Imprimatur: Philip Shirvani and Nirmal Saxena

List of Figures

List of Tables

1. Introduction

Errors in computer systems can cause abnormal behavior and degrade system reliability, data integrity and availability. This is especially true in a space environment where transient errors are a major cause of concern. Fault avoidance techniques such as radiation hardening and shielding have been the main approaches to meet the reliability requirements. Recently, unhardened *Commercial Off-The-Shelf* (COTS) components have been investigated for space applications because of their higher density, faster clock rate, lower power consumption and lower price. Since COTS components are not radiation hardened, and it is desirable to avoid shielding, *Software-Implemented Hardware Fault Tolerance* (SIHFT) was proposed to increase the data integrity and availability of COTS systems.

Error Detection by Data Diversity and Duplicated Instructions (ED^4I) is a SIHFT technique that detects both permanent and temporary faults by executing two "different" integer programs with different data sets (with the same functionality) and comparing their outputs. The "different" programs with diverse data are generated in the following way. Given an integer program that contains integer variables and constants, we automatically transform it into a new program in which all integer variables and constants are multiplied by a *diversity factor k* that is an integer constant. Depending on the factor *k*, the original and the transformed program may use different parts of the underlying hardware and propagate fault effects in different ways; therefore, if the two different programs produce different outputs due to a fault, we can detect the fault by examining whether the results of the transformed program are also *k* times greater than the results of the original program. There are two ways to check the results. First, another concurrently running program can compare the results. Second, the main program that spawns the original program and the transformed program checks their results after they are completed.

The value of the factor *k* determines the error detection capability of ED^4I; it should satisfy two goals. The primary goal is to guarantee *data integrity*; that is, the probability that the two programs do not produce identical erroneous outputs. The secondary goal is to maximize the probability that the two programs produce different outputs for the same hardware fault so that error detection is possible (error detection probability). However, the factor *k* should not cause an overflow in the functional units. In order to determine the optimum value of *k*, we have developed an analysis technique based on the probabilistic modeling of logic networks [Parker 75] and design *diversity metric* [Mitra 99]. The diversity metric was used in [Mitra 99] to quantify diversity among several designs. We use this metric to measure the diversity between the original and transformed programs.

We have implemented a preprocessor that automatically transforms a program to a new program based on the algorithm described in Appendix I. The flow is illustrated in Fig. 1.1.

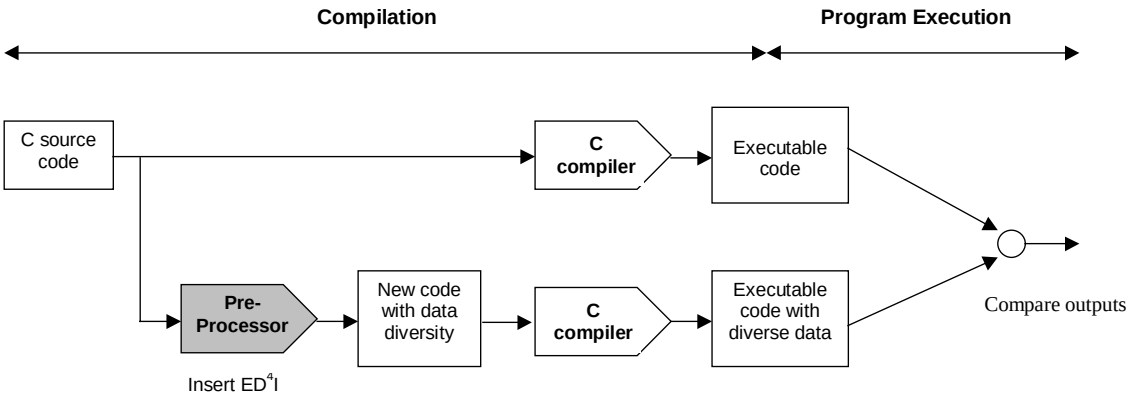


Fig. 1.1. Application procedure for ED^{4}I.

Unlike the previous data diversity techniques [Avizienis 77][Scot 83][Laprie 90] that target software faults, this paper presents a new approach to data diversity for on-line hardware fault detection. Our contributions are: (1) devising an algorithm that transforms a program to a new program with diverse data, (2) quantifying diversity between programs using a metric that was previously developed for hardware diversity, (3) presenting a probabilistic analysis technique and closed form solutions to determine the optimum value of *k*, (4) performing simulation experiments to illustrate how to choose *k*, and (5) showing how system availability is increased by our technique. We discuss previous work in Sec. 2 and present program transformation algorithm in Sec. 3. We discuss how to handle overflow in Sec. 4 and present how to determine an optimal value of *k* in Sec. 5. We show benchmark simulation results in Sec. 6, and explain how to handle floating point numbers in Sec.7. Finally, conclusions are presented in Sec 8.

2. Previous Work

Design diversity has been proposed in the fault tolerance literature to increase the reliability of the system. Design diversity is defined as the independent generation of two or more different software or hardware elements to satisfy a given requirement [Avizienis 84]. The main objective of design diversity is to protect redundant system from c*ommon-mode failures*, which are failures that affect more than one module at the same time [Lala 94]. Design diversity also has been applied to software systems [Lyu 91]. *N version programming* (NVP) [Avizienis 77][Chen 78][Avizienis 85] is one example of diversity in software. Design diversity in N version

programming targets software design faults. In N version programming, different designers develop independent versions of the programs to avoid common design errors. The *Consensus recovery block* technique [Scot 83][Scot 87] is a hybrid system combining NVP and recovery blocks: It is another example in which diversity is used in software. A variant of NVP, *N self-checking programming,* also employs the design diversity concept and is used in the Airbus A310 system [Laprie 90][Dugan 93].

In [Ammann 88][Christmansson 94], data diversity was proposed for detecting software faults. In [Ammann 88], input data in unused input space (input data that are not used during normal execution of the program) are applied to detect faults in software, and in [Chrismansson 94], diverse data from replicated sensors are used to tolerate faults in software.

However, our technique, ED^{4}I, is different from the previous software diversity or data diversity techniques. First, our target is not software design faults but hardware faults – both permanent and transient faults in the system. Second, while the previous techniques preserve the original program structure and apply diverse data in different or unused input domain to the original program, we transform a program to a new program in which the data are automatically diversified using a transformation algorithm presented in Sec. 3.

In the transformed program, the values of all integer variables and constants are multiplied by the diversity factor k in our technique. This is similar to the *AN code* [Brown 60] in which each data word is multiplied by some constant A, but the error detection method in our technique is different from that in the *AN* code. The *AN* code detects an error by checking if the computation result is divisible by A. By contrast, our technique detects an error by comparing the results of the original and the transformed program (not by checking whether the result is divisible by A or not). Our technique is also different from Recomputing with Shifted Operands (RESO) presented in [Patel 82]. In RESO, the underlying hardware is modified so that each operation is recomputed with shifted operands. However, ED^{4}I is a SIHFT technique that does not change the original hardware. Instead, a "different" program is created by the transformation, and comparing outputs of the original and the new program detects errors.

[Engel 97] suggests modifying the program so that all variables are negated. This is the same as using -1 for the value of k. However, our results show that the value of $k = -1$ is not the optimum value because for some functional units, data integrity is not guaranteed for $k = -1$. This will be shown in Sections 5 and 6. By contrast, our technique quantifies diversity and chooses the optimum diversity factor k (not limited to only -1) to maximize data integrity as well as fault detection capability. We describe the algorithm formally in Sec. 3 and prove the correctness in Appendix I.

3. Program Transformation

This section presents a transformation algorithm that transforms a program to a new program with diverse data. We will begin with definitions of terminologies, and then describe the transformation algorithm. Finally, we will show an example illustrating how the transformation is performed.

A *basic block* is *a sequence of consecutive statements in which the flow of control enters at the beginning and leaves at the end without branching except at the end.* By defining $V = \{v_1, v_2,\ldots,v_n\}$ as *the set of vertices representing basic blocks*, and $E = \{(i,j) \mid (i,j)$ is a branch from v_i to $v_j\}$ as *the set of edges denoting possible flow of control between the basic blocks*, a program can be represented by a *program graph*, $P_G = \{V, E\}$. For example, for the program in Fig. 3.1 (a) (the corresponding flow graph is shown in Fig. 3.1 (b)), there are four basic blocks: v_1, v_2, v_3 and v_4 as shown in Fig. 3.1 (c). Therefore, $V = \{v_1, v_2, v_3, v_4\}$, $E = \{(1,2), (2,3), (3,2), (2,4)\}$.

If x is k times greater than y, x is *k-multiple* of y. *Program transformation* transforms a program P to a new program P' with diverse data in which all variables and constants are k-multiples of the original values when the program P' is executed. It consists of two transformations: *expression transformation* and *branching condition transformation.* The expression transformation changes the expressions in P to new expressions in P' so that the value of every variable or constant in the expression of P' is always the k-multiple of the corresponding value in P. Since the values in P' are different from the original values, when we compare two values in a conditional statement, the inequality relationship might need to be changed. For example, the conditional statement `if (i < 5)` in P needs to be changed to `if (i > -10)` in P' when k is -2. Otherwise, the control flow determined by the conditional statements in P' would be different from the control flow in P, and the computation result from the diverse program P' would not be the k-multiple of the result from the original program. The branching condition transformation adjusts the inequality relationship in the conditional statement in P' so that the control flows in P and P' are identical.

A *k-factor diverse program* is *a program with a new program graph $P_G' = \{V', E'\}$ that is isomorphic to P_G, but all the variables and constants in P' are k-multiples of the ones in P.*

Let us define S and S' as the sets of variables in P and P' respectively, and define n as the number of vertices (basic blocks) executed; then, $S(n)$ and $S'(n)$ are defined as:

$S(n)$: *the set of values of the variables in S after n basic blocks are executed,*

S´(n): the set of values of the variables in S´after n basic blocks are executed.

For example, in Fig.3.1, the set of the variables in the program is S = {`i, x, y, z`}. After the program is started and one basic block is executed, n = 1 and $S(1)$ = {`i=0, x=1, y=5, z=0`} because the four statements in the first basic block are executed. After v_2 and v_3 are executed, n = 3 and $S(3)$ = {`i=1, x=1, y=5, z=1`}.

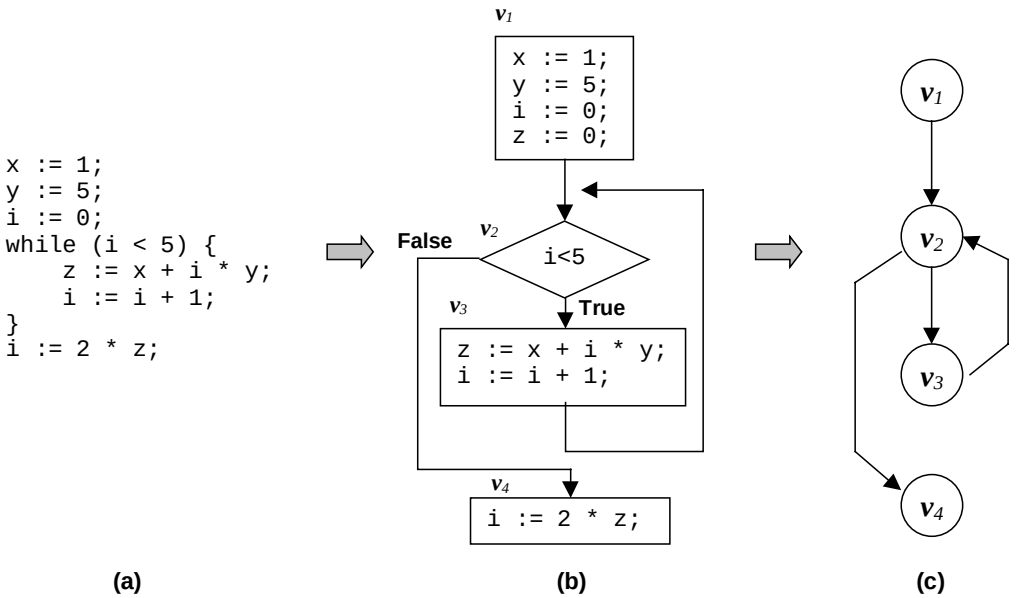


Fig.3.1. (a) An example program (b) A flow graph (c) A program graph P_G

The program transformation should satisfy:

(1) P_G and P_G´are isomorphic.

(2) $k \cdot S(n) = S'(n)$, for $\forall n > 0$

(where $k \cdot S(n)$ is obtained by multiplying all elements in $S(n)$ by k).

The condition in (1) tells us that the control flow in the two programs should be identical. The condition in (2) requires that all the variables in the transformed program are always k-multiples of those in the original program.

In the expression transformation, we build a parse tree for every expression in P, and produce a new expression by recursively transforming the parse tree. In the branching condition transformation, we examine the inequality relationship in the conditional statements and modify it according to the value of k. Then, the transformed program always satisfies the conditions stated in (1) and (2). The formal description for the program transformation algorithm as well as the correctness and proofs of the algorithm is presented in Appendix I.

The sample program in Fig.3.1 is transformed to a diverse program and shown in Figs. 3.2 and 3.3 where k = -2.

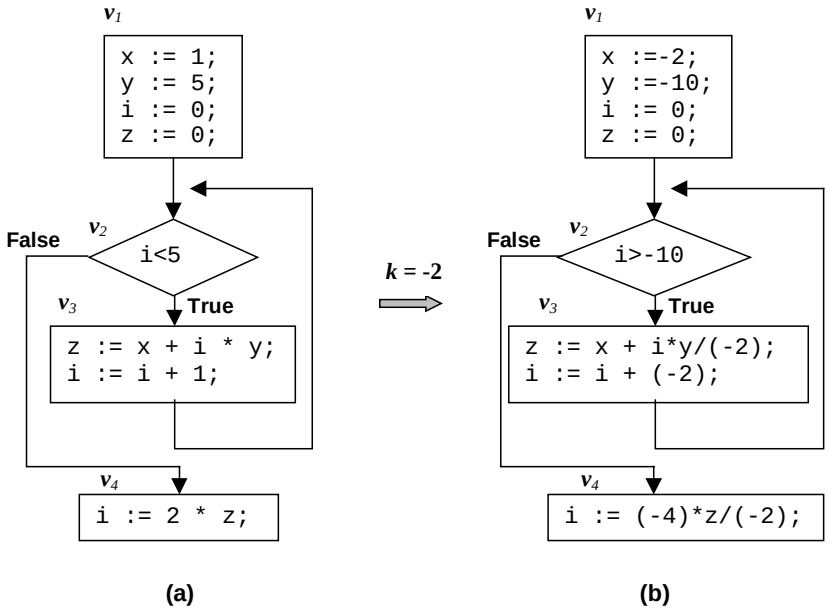


Fig.3.2. (a)The original program (b) The transformed program with k = -2.

```
x := 1;
y := 5;
i := 0;
while (i < 5) {
    z := x + i * y;
    i := i + 1;
}
i := 2 * z;
```

(a)

k = -2

```
x := -2;
y := -10;
i := 0;
while (i > -10) {
    z := x + i * y /(-2);
    i := i + (-2);
}
i := (-4) * z / (-2);
```

(b)

Fig. 3.3. (a) The original program (b) The transformed program with k = -2

There are limitations in applying ED^{4}I to floating point number arithmetic; this issue will be discussed in Sec. 7. Also we assume that the factor k is determined to cause no overflow when the transformed program is executed. The factor k can be any integer if the program contains only integer arithmetic operations, but k should be 2^l (where l is an integer) if the program includes logical operations such as bit-wise AND, OR, or XOR.

4. Overflow Handling

The primary cause of the overflow problem is the fact that in the transformed program (after multiplication by *k*), the size of the resulting data may be too big to fit into the data word size of the processor. For example, consider an integer value of 2^{31}-1 in a program (with 32-bit 2's complement integer representation). If the value of *k* is 2, then the resulting integer (2^{32}-2) cannot be represented using 32-bit 2's complement representation. Also, consider a floating point number $1.5 \times 2^{2^8-1}$ with 8-bit exponent representation. If the value of *k* is 2, then the resulting floating number 1.5×2^8 cannot be represented using 8-bit exponent representation. Note that, even if *k* = -1, this overflow problem will occur (e.g., consider the data value of -2^{32}; multiplication by -1 means a data value of 2^{32} which cannot be represented in 32-bit 2's complement representation). Previous hardware techniques like RESO [Patel 82] eliminated this overflow problem by adding extra bit-slices in the datapath. In SIHFT techniques such as ED^{4}I, hardware changes are avoided so that these techniques can be used with COTS components.

There are three ways to handle this overflow problem.

4.1. Scaling

The overflow problem can be solved by two scaling techniques: scaling up to higher precision and scaling down the original data. In the first solution, we scale up the data type in the program to avoid overflow; data types such as 16-bit single-precision integers can be scaled up to 32-bit double-precision integer data type. Thus, modification of the program is required. However, in the second solution, we scale down the original input data to avoid overflow and do not modify the program.

If we have an overflow in one data type that has *N* bits, then we can scale up the data type so that the data type will have more than *N* bits. For example, suppose integers are represented as 16 bit words in a particular system. If we have an overflow in a variable of this integer type, we can change the type of the variable to double precision integer data type that uses 32-bit words. In this way, we can avoid overflow problems. Similarly, for double precision integer variables, there is a possibility that we can have an overflow. If there is no data type that can represent more than 32-bit double-precision integers in the system, then we can combine two 32-bit double-precision words to form a 64-bit multiple-precision integer word and change the program appropriately.

As an example, in Fig. 4.1, 64-bit integer x can be represented by combination of two 32-bit double-precision integers: x_l representing the lower 32 bits of x and x_h representing the higher 32 bits of x. Suppose y is also represented by y_l and y_h , and z is represented by z_l and z_h. When we add two numbers x and y, we need two additions; addition of lower bit words and addition of higher bit words. A pseudo code for this addition is:

```
z = x + y  → add zl, xl, yl     ; the first addition for lower bit word
             addc zh, xh, yh    ; the second addition for higher bit word
```

In the code, `addc` is an addition with a carry bit generated by the previous addition.

The first addition in (a) reads two lower bit words (x_l and x_h) from registers, performs an addition and stores a result back in the lower bit word of the result z. The second addition in (b) reads two higher bit words (y_l and y_h) from the registers, performs an addition with the carry generated from (a), then stores the result in the higher bit word of the result z.

Similarly, scaling up the data precision requires modification of other arithmetic computations such as subtraction, multiplication and division. In these computations, lower bit words and higher bit words should be computed separately. We need to calculate lower bit words first, subsequently, calculate higher bit words with the result from lower bit word computation result, and then combine the two computation results to produce one word. This modification should be done in all arithmetic computations throughout the entire program.

Separate calculation of lower bit and higher bit words requires more execution time, resulting in performance overhead. However, scaling up to higher precision data type will remove the overflow problem during the execution of the program.

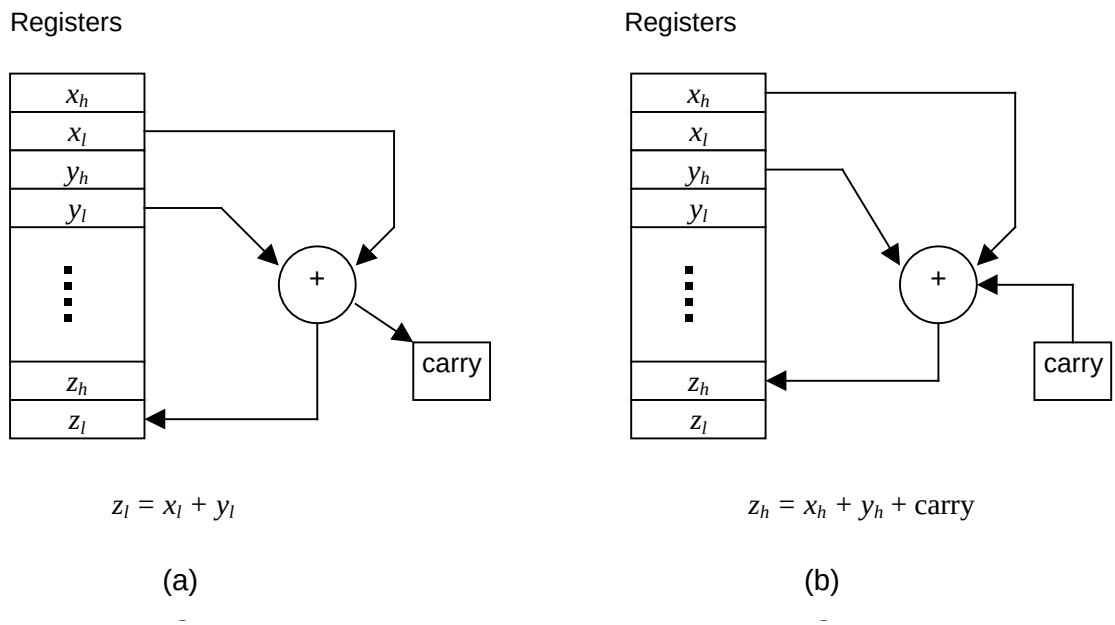


Fig. 4.1. (a) Lower bit word addition (b) Higher bit word addition with a carry

In the second solution, we can scale down the data before the execution of the original program in order to avoid overflow. For example, suppose the input data ranges from 2^{20} to 2^{31} in a sorting program, but only 32-bit integers (2^{32}) are allowed in our system. If the most significant bit is a sign bit, we cannot use $k > 1$. In this case, we can scale down data in the transformed program. For example, we can use $k = -2^{-2}$ so that we can avoid overflow in the transformed program. There is a possibility that scaling down data may cause computation inaccuracy during the execution of the program. Therefore, when we compare the scaled down values with the original values, we have to compare the higher order bits that are not affected by scaling down.

4.2. Data range check at compile time

Modern compiler techniques enable us to estimate the range of values for each variable in a program. Harrison [Harrison 77] presents a technique to determine bounds on the ranges of values assumed by variables at various points in the program. Patterson [Patterson 95] described using value range propagation for accurate static branch prediction. Stephenson [Stephenson 00] presents a compiler technique that minimizes the bitwidth of registers by finding the minimum number of bits needed to represent each program operand. This data range analysis can be applied to find a maximum bound of variables in the program and used to determine a value k that does not cause an overflow, if such k exists.

4.3. Special error handlers during run-time

In many microprocessors such as the Intel Pentium, special status flags (e.g., overflow, carry out, etc.) are reported at the end of each arithmetic operation. If the comparison of the original program and the transformed program produce a mismatch, then an error handler must be invoked for recovery. Special functions can be written inside the error handler to check the overflow bit of the status register. If a mismatch occurs and the overflow bit is not set (assuming no error occurred in the status flags), the error handler knows that the mismatch is due to a fault and invokes a recovery process on the assumption that an error has occurred during execution. On the other hand, when a mismatch occurs and the overflow bit is set, the error handler knows that the mismatch is due to an overflow and requests an overflow recovery. The overflow recovery process can execute another version of the transformed program with a different value of k that is less than the previous value. In this case, we have assumed that several versions of the transformed program with different values of k are available in the system. We have to consider the case when an error occurs in data and the overflow bit is also erroneously set. In this case, the error handler will report an overflow and re-execute the programs. If the data error and overflow error were transient, re-execution of the programs will produce correct results. If only the overflow bit error was transient, re-execution of the program will detect the permanent fault in the data path. If the overflow bit was permanently stuck at 1, every computation will report an overflow. These repeated overflow indications will be caught by the operating system. A flowchart showing decision steps is illustrated in Fig. 4.2.

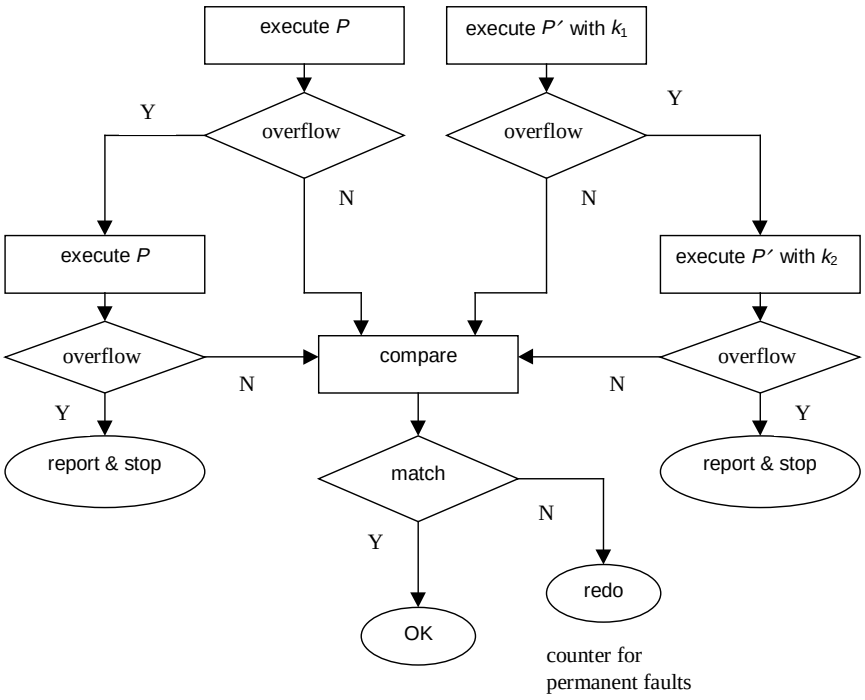


Fig. 4.2. A flowchart for the determination of overflow and errors.

5. Determination of *k*

The factor *k* determines how diverse the transformed program is. This section considers how to choose an optimal value of *k* that maximizes the diversity of the transformed program. For this purpose, we have developed two metrics to measure the diversity of the transformed program: data integrity and fault detection probability. An optimal value of *k* is the value that satisfies two goals: the primary goal is to maximize the data integrity and the secondary goal is to maximize the fault detection probability. Data integrity is more important because it guarantees no undetected error. For any given program, we first analyze the data integrity and fault detection probability of each functional unit of the system for various values of *k*. Next, as described in Sec.5.6, we use these values to create an optimal value of *k* for the transformed program by looking at execution profiles of the programs.

Sec. 5.1 presents a diversity metric we have adopted from [Mitra 99], in which a diversity metric for systems with redundant hardware has been developed. From Sec. 5.2 to Sec. 5.5, we analyze data integrity and fault detection probability in functional units: bus, adders, a multiplier, and a shifter. In our analysis, we consider integers from –5 to 5 for the value of *k.* Integers whose

absolute values are greater than 5 are not considered for k in this paper because they have higher probability of overflow than the values considered. The analyzed data integrity and fault detection probability for functional units will be used in the next section to determine the optimum value of k for benchmark programs using execution profiles of the programs.

5.1. Diversity Metric: Fault Detection Probability & Data Integrity

Researchers have studied techniques to quantify diversity in multiple designs and which technique should be used to measure the diversity [Eckhardt 85][Litt 89][Lyu 91]. They use random variables Π and X to represent arbitrary programs and arbitrary inputs, and the probability that Π fails on X is calculated. Our diversity metric is somewhat different because we need to compute the probability that a program fails due to a hardware fault in the system.

In our approach, fault detection probability and data integrity quantify diversity between two programs running on the same hardware. Let us define:

- X *the set of corresponding inputs to a functional unit h_j in the system when an integer program P executes*
- X' *the set of corresponding inputs to a functional unit h_j in the system when an integer program P′ executes*
- x *a particular input to h_j produced by P and $x \in X$*
- x' *a particular input to h_j produced by P′ and $x' \in X'$*
- $|X|$ *the number of elements in the set X*

Then, the output y of h_j with input x and the output y' of h_j with input x' should satisfy the relationship $y' = k \cdot y$ unless a fault occurs in h_j without any overflow. In the presence of a fault f_i in h_j, let us define:

- E_i *the subset of X that contains inputs producing incorrect outputs in h_j*
- E_i' *the subset of E_i that contains inputs producing incorrect outputs that erroneously satisfies the relationship $y' = k \cdot y$ in the presence of a fault f_i*

Then, $|E_i - E_i'|$ is the number of incorrect outputs that have the relationship $y' \neq k \cdot y$ and are detected the fault by mismatch.

We define the *fault detection probability in h_j, $C_j(k)$*, as:

$$(3) \qquad C_j(k) = \sum_i \Pr\{f_i\} \Pr_i\{y \neq ky\} = \sum_i \Pr\{f_i\} \left(\frac{|E_i - E_i'|}{|X|} \right).$$

Moreover, we define the *data integrity in h_j, $D_j(k)$*, as:

$$D_j(k) = \sum_i \Pr\{f_i\}\left(1 - \frac{|E_i'|}{|X|}\right) \tag{4}$$

If we assume a uniform distribution for all faults,

$$\begin{aligned} C_j(k) &= \sum_i \Pr\{f_i\}\left(\frac{|E_i - E_i'|}{|X|}\right) = \sum_i^{N_f} \frac{1}{N_f}\left(\frac{|E_i - E_i'|}{|X|}\right) \\ D_j(k) &= \sum_i^{N_f} \Pr\{f_i\}\left(1 - \frac{|E_i'|}{|X|}\right) = \sum_i^{N_f} \frac{1}{N_f}\left(1 - \frac{|E_i'|}{|X|}\right) \end{aligned} \tag{5}$$

where N_f denotes the total number of faults in h_j.

For various values of k, $C_j(k)$ and $D_j(k)$ of h_j can be obtained by either probabilistic method or simulation method. They are weighted by execution frequency u_j of h_j in the program execution profile and summed to obtain the overall $C(k)$ and $D(k)$ for a particular program.

$$\begin{aligned} C(k) &= \sum_j u_j C_j(k) \\ D(k) &= \sum_j u_j D_j(k) \end{aligned} \tag{6}$$

An optimal k is the value that maximizes $C(k)$ with the highest $D(k)$.

From Sec.5.2 to Sec.5.5, we show $C_j(k)$ and $D_j(k)$ in functional units; then, in Sec. 5.6, we show how to determine optimal values of k for each benchmark program.

5.2. Bus Signal Line

An M bit bus consists of M parallel signal paths. As shown in Fig. 5.1, the source places an M bit signal x on the M bit bus. If the ith bit of the bus has a stuck fault, the destination may receive a corrupted x. If there is a parity bit on the bus, a single bit error can be detected by parity check. However, if there is no parity on the bus, the fault on this bus can be detected by the ED^{4}I technique.

If we put x of the original program and x' (= kx) of the diverse program on the bus one after the other, the fault will affect x and x' in different ways. If the fault corrupts either x or x', but not both, the relationship $x' = k \cdot x$ in the destination will not be satisfied. On the other hand, if the fault corrupts both x and x' in different ways, the relationship $x' = k \cdot x$ in the destination may or may not be satisfied. If neither x nor x' provoke the fault, the destination will receive correct values and also satisfy $x' = k \cdot x$. In this case, the fault does not corrupt the information.

If k is –1 or a power of 2, we can get closed form solutions for the data integrity and fault detection probability. The closed form solutions are derived in Appendix II and summarized in Table 1. Table 2 shows fault detection probability $C_j(k)$ and data integrity $D_j(k)$ for different

values of k in a 12 bit (M = 12) bus as an example. A simulation method applying exhaustive input patterns in the presence of faults is used when k is not –1 nor a power of 2. In the table, negative numbers are represented in a 2's complement representation that is widely used in most microprocessors.

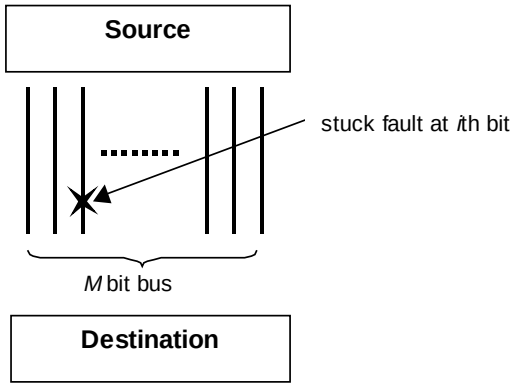


Fig. 5.1. An M bit bus transferring information from a source to a destination unit.

Table 1. Closed form solutions for $C_j(k)$, $D_j(k)$ in the presence of a fault in the ith bit of M-bit bus.

	$k = -2^l$ ($l > 1$)	$k = -2$	$k = -1$	$k = 2$	$k = 2^l$ ($l > 1$)
$C_j(k)$	$\frac{3}{4}+\frac{1}{2^i}(1-\frac{1}{2^{l-1}})-\frac{1}{2^M}$	$\frac{3}{4}-\frac{1}{2^M}$	$1-\frac{1}{2^{i+1}}$	$\frac{3}{4}$	$\frac{3}{4}+\frac{1}{2^{i+1}}(1-\frac{1}{2^{l-1}})$
$D_j(k)$	1	1	1	1	1

Table 2. $C_j(k)$ and $D_j(k)$ in a 12 bit bus (M = 12). Equation (5) is used to calculate values.

	k = -5	k = -4	k = -3	k = -2	k = -1	k = 2	k = 3	k = 4	k = 5
$C_j(k)$	0.7433	0.7780	0.7593	0.7500	0.9167	0.7500	0.6574	0.7656	0.6733
$D_j(k)$	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

In Table 2, the highest fault detection probability occurs when k = -1. The value –1 for k has an advantage that the chance of overflow is very low. On the other hand, $C(k)$ is larger when $|k|$ is 2 and 4 compared to the cases when k is an odd number such as 3 and 5. Therefore, the greatest value of $k = 2^l$ might be the best choice for the lowest performance overhead (as long as an overflow does not occur) because multiplication by 2^l can be replaced by l-bit shifting operation that usually takes one cycle in most microprocessors.

5.3. Adder

An iterative logic array such as a ripple carry adder consists of multiple logic cells cascaded in series. It receives carries from a previous cell and inputs to this cell; then, it produces outputs and subsequently generates a carry to the next cell.

The regularity of the array helps us to calculate the fault detection probability when the multiplying factor k is 2^l, where l is an integer. Suppose one node of the cell s_i has a stuck fault. If x is an output of the array, the ith bit of x may be corrupted. In the transformed program, if $x' = kx$ is the output of the array, the ith bit of x' may be corrupted; however, this is equivalent to that the i-lth bit of x is corrupted because x' is an l bit shifted x. Therefore, we can calculate the probability of provoking the faults in two cells s_{i-l} and s_i, and get the probability of mismatch between x and x', i.e., $\Pr\{k \cdot x \neq x'\}$.

This section considers adders as an example of an iterative logic array. We analyze the probability that a signal changes from 0 to 1 and from 1 to 0 in every node in the adder and compute the probability of provoking the faults using the method in [Parker 75]. This analysis gives us the probability $C_j(k)$. The details of the analysis technique are described in Appendix II.

For every single stuck-at fault f in a ripple carry adder and a carry look-ahead adder, we calculated the values of $C_j(k)$ and $D_j(k)$ using our analysis technique and exhaustive simulation of all possible input combinations with various values of k. The numbers are reported in Table 3 and Table 4.

As shown in Table 3 and 4, we cannot achieve data integrity of 1 when k = -1; this explains why our technique is better than [Engel 97] in which all variables are negated (same as k = -1 in our technique). If k is -1, there are some faults in the adder that will not be detected. For example, suppose one of the XOR gate's output is stuck at 0 in a full adder of the ith stage s_i of the adder (Fig. 5.2). If the input **a** of the adder is 2^i, and the other input **b** is 1, this fault cannot be detected as described in Fig. 5.2 (b). From this observation, we can see that the value -1 is not suitable for k in terms of data integrity.

For the same absolute value of k, a negative k has higher fault detection probability than a positive k. Multiplying by an odd number such as 3 is not as efficient as multiplying by 2, which is just a one bit shifting operation. Therefore, we do not need to choose 3 for k, which is a more expensive operation than shifting by just one bit.

Among the values shown in the table, k = -2 shows the highest fault detection probability under the condition that data integrity is 1.

Table 3. $C_j(k)$ and $D_j(k)$ in a 12 bit ripple carry adder with various k values

	k = -5	k = -4	k = -3	k = -2	k = -1	k = 2	k = 3	k = 4	k = 5
$C_j(k)$	0.643	0.657	0.639	0.662	0.665	0.594	0.563	0.612	0.557
$D_j(k)$	1.000	1.000	1.000	1.000	0.951	1.000	1.000	1.000	1.000

Table 4. $C_j(k)$ and $D_j(k)$ in a 12 bit carry look-ahead adder with various k values

	k = -5	k = -4	k = -3	k = -2	k = -1	k = 2	k = 3	k = 4	k = 5
$C_j(k)$	0.609	0.620	0.603	0.629	0.639	0.559	0.537	0.578	0.522
$D_j(k)$	1.000	1.000	1.000	1.000	0.963	1.000	1.000	1.000	1.000

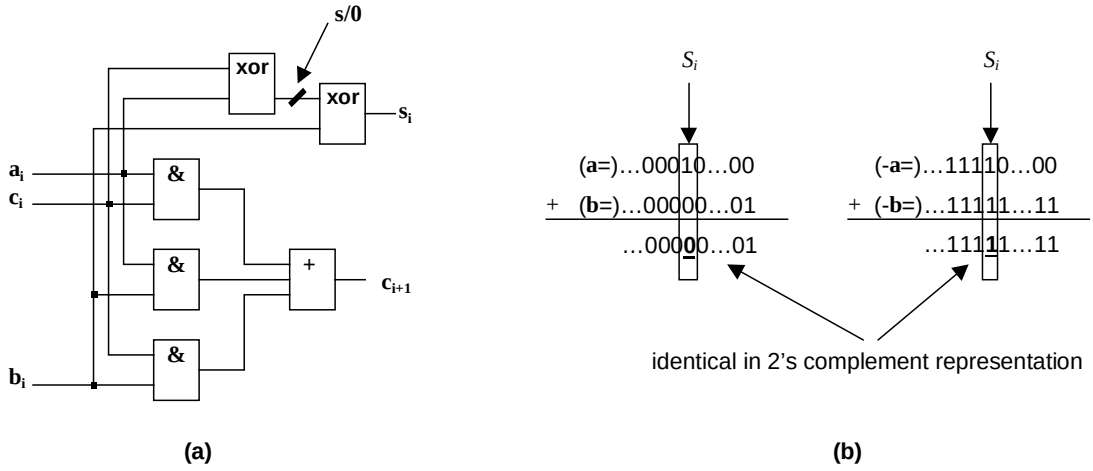


Fig. 5.2. (a) s/0 at the output of XOR gate in S_i, the ith cell of a ripple-carry adder.
(b) The s/0 fault shown in (a) cannot be detected by comparing the two values although we have erroneous computation results.

5.4. Multiplier

Many different implementations for multipliers exist, and data integrity and fault detection probability depend on the particular design implementation. In this section, we consider a parallel array multiplier [Weste 92] to demonstrate the dependence of $C_j(k)$ and $D_j(k)$ on various values of k. The parallel array multiplier has a regular cell structure as shown in Fig. 5.3 in which a 4-bit multiplier is illustrated.

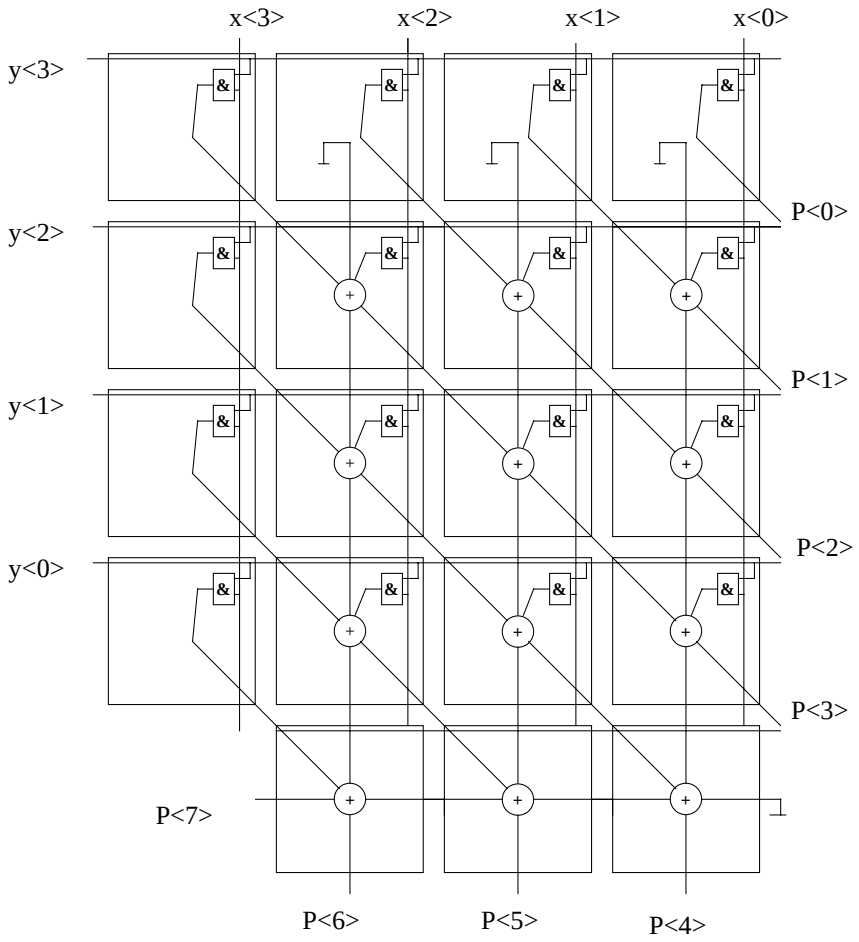


Fig. 5.3. A 4-bit parallel array multiplier.

In a 12 bit array multiplier as an example, we randomly selected a node in the multiplier and injected a stuck fault into the node. We injected a fault randomly, applied exhaustive input patterns to the multiplier in the presence of a fault and repeated simulations 10^4 times. We show the result in Table 5.

The table shows only positive values of k because the array multiplier treats negative number multiplication as a positive number multiplication after changing the sign of the negative numbers.

The table shows us that the highest $C_j(k)$ in the table occurs when $k = 4$. Note that, when $k = -1$, $C_j(k) = D_j(k) = 0$ because the array multiplier converts the transformed program's negative numbers to original positive numbers before multiplication. This results in the same multiplication in the original and transformed program, and consequently, the fault cannot be detected.

Table 5. $C_j(k)$ and $D_j(k)$ in a 12 bit parallel array multiplier

	$k = 2$	$k = 3$	$k = 4$	$k = 5$
$C_j(k)$	0.5978	0.5851	0.5997	0.5887
$D_j(k)$	1.0000	1.0000	1.0000	1.0000

5.5. Shifter

Since multiplication or division by a power of 2 can be replaced by shifting operation, shifters are used frequently during program execution [Weste 92]. Because a multiplexer-based shifter as shown in Fig. 5.4 [Weste 92] is widely used, we will take this design for our simulation.

In our simulation, we injected a stuck-at fault into a randomly selected node and applied exhaustive patterns to 16 bit shifter inputs. This simulation experiment was repeated for 10^4 times with a randomly chosen single stuck-at faults. The results are shown in Table 6.

In this shifter implementation, Table 6 shows that the value of -1 for k has the highest fault detection probability with guaranteed data integrity.

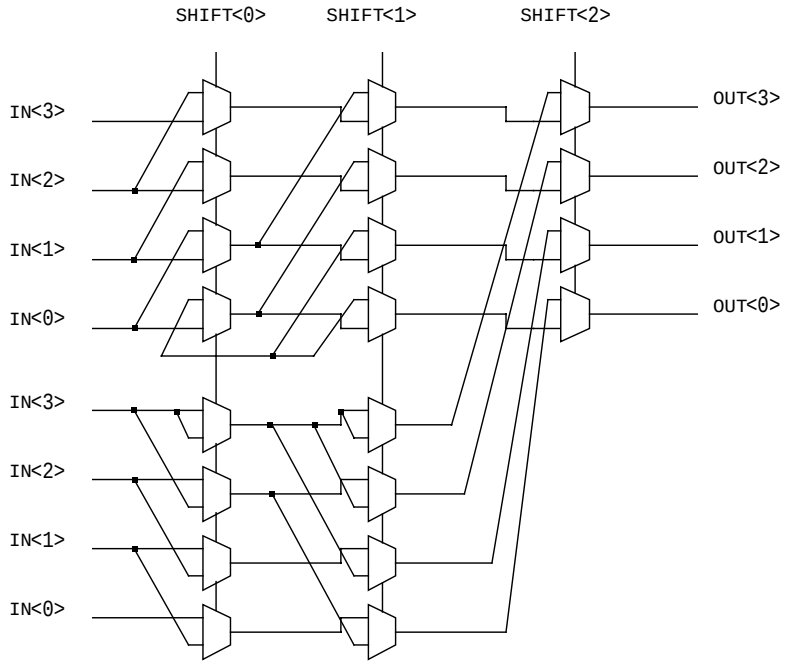


Fig. 5.4. A 4-bit multiplexer-based shifter.

Table 6. $C_j(k)$ and $D_j(k)$ in a 16 bit multiplexer-based shifter

	$k = -5$	$k = -4$	$k = -3$	$k = -2$	$k = -1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$
$C_j(k)$	0.3572	0.3685	0.3722	0.3754	0.4228	0.3198	0.2900	0.3124	0.2927
$D_j(k)$	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.9963	1.0000	0.9963

5.6. Determination of k for benchmark programs

We have observed that in different functional units, we have different values of k that maximize the fault detection probability and data integrity. For example, the bus has the highest fault detection probability when k = -1, but the array multiplier has the highest fault detection probability when k = 4. Therefore, programs, such as matrix multiplication that use a multiplier extensively, will need 4 or -4 for the value of k to obtain the highest fault detection probability. However, programs such as sorting that use memory buses heavily to communicate with memory for loading and storing data need –1 for the value of k to maximize its fault detection probability. Hence, an execution profile of a program showing execution frequencies of each functional unit is necessary to determine the best k for a particular program.

Table 7 shows execution profiles obtained from simulating several benchmark programs in a MIPS simulator. The simulator gives us the execution frequency of each functional unit when benchmark programs are executed. The first three entries (add/sub, multiplication, and shift) in the table represent the execution frequencies of instruction types that use an adder, a multiplier, and a shifter relatively. The fourth entry, a memory access type, is an operation that mainly uses the memory bus. For a branch instruction type, we assume that a carry-look ahead adder is mainly used. The rest of the instruction types are included in the entry of the last row. In Table 8 and Table 9, $C(k)$ and $D(k)$ are calculated from equation (6), which uses the values of $C_j(k)$ and $D_j(k)$ from Table 1 to Table 6.

An optimal value of k must satisfy the primary and the secondary goals. The primary goal is to maximize the data integrity and the secondary goal is to maximize the fault detection probability. Table 8 shows the values of k that satisfies the primary goal of maximizing data integrity. Those values are indicated by shaded entries in the table. Under the condition that this primary goal is satisfied, Table 9 shows the value of k that maximizes the fault detection probability (indicated by shaded entry in the table). Therefore, the value in the shaded entry in Table 9 is an optimum k for each benchmark program.

Although the highest fault detection probability occurs when k = -1 in six benchmark programs, the value of –1 is not a good choice for k because the data integrity is the lowest when k = -1. In those benchmark programs, -2 is the best choice for k because the fault detection probability is maximized (shaded entries in Table 9) under the condition that data integrity is the highest (shaded entries of Table 8).

Table 7. Execution frequencies of instruction types in bench mark programs.

	I-sort	Q-sort	Lzw	Fib	Mat-mul	Shuffle	Hanoi
add/sub	50.7%	50.7%	44.8%	46.4%	51.1%	42.3%	33.7%
multiplication	0%	0%	0%	0%	10.0%	1.0%	0%
shift	5.2%	6.3%	2.5%	0%	1.8%	33.7%	4.0%
memory access	21.6%	23.5%	32.7%	28.6%	29.8%	19.3%	51.5%
branch	22.5%	19.5%	19.4%	25.0%	7.3%	2.7%	10.8%
others	0%	0%	0.6%	0%	0%	1.0%	0%

Table 8. Data Integrity *D(k)* calculated with various values of *k* in benchmark programs. Note that Shaded areas indicate the highest data integrity in a row.

	k = -5	*k* = -4	*k* = -3	*k* = -2	*k* = -1	*k* = 2	*k* = 3	*k* = 4	*k* = 5
I-sort	1.0000	1.0000	1.0000	1.0000	0.9641	1.0000	0.9998	1.0000	0.9998
Q-sort	1.0000	1.0000	1.0000	1.0000	0.9656	1.0000	0.9998	1.0000	0.9998
Lzw	0.9940	0.9940	0.9940	0.9940	0.9625	0.9940	0.9939	0.9940	0.9939
Fib	1.0000	1.0000	1.0000	1.0000	0.9650	1.0000	1.0000	1.0000	1.0000
M-mul	1.0000	1.0000	1.0000	1.0000	0.8714	1.0000	0.9999	1.0000	0.9999
Shuffle	0.9900	0.9900	0.9900	0.9900	0.9580	0.9900	0.9888	0.9900	0.9887
Hanoi	1.0000	1.0000	1.0000	1.0000	0.9782	1.0000	0.9999	1.0000	0.9998

Table 9. Fault detection probability *C(k)* calculated with various values of *k* in bench mark programs. Shaded areas indicate the highest fault detection probability under the condition that data integrity is the highest (shaded areas in Table 8).

	k = -5	*k* = -4	*k* = -3	*k* = -2	*k* = -1	*k* = 2	*k* = 3	*k* = 4	*k* = 5
I-sort	0.6491	0.6621	0.6511	0.6661	0.7068	0.6134	0.5692	0.6262	0.5837
Q-sort	0.6479	0.6607	0.6505	0.6646	0.7089	0.6134	0.5680	0.6256	0.5824
Lzw	0.6642	0.6763	0.6678	0.6796	0.7373	0.6346	0.5837	0.6484	0.5986
Fib	0.6710	0.6836	0.6734	0.6872	0.7370	0.6386	0.5900	0.6536	0.6053
M-mul	0.6617	0.6813	0.6647	0.6766	0.6691	0.6359	0.5884	0.6480	0.6023
Shuffle	0.5586	0.5705	0.5654	0.5751	0.6187	0.5258	0.4838	0.5328	0.4946
Hanoi	0.6699	0.6934	0.6769	0.6823	0.7679	0.6519	0.5903	0.6638	0.6051

Table 10. Optimum value of *k* determined for each benchmark programs.

	I-sort	Q-sort	Lzw	Fib	Mat-mul	Shuffle	Hanoi
Optimum *k*	-2	-2	-2	-2	-4	-2	-2

During the execution of the six benchmark programs (I-sort, Q-sort, Lzw, Fib, Shuffle, and Hanoi), the most frequently used functional units are adders. A carry-look ahead adder and a ripple carry adder have the highest fault detection probability (under the condition that data integrity is the highest) when k = -2; thus, k = -2 is also the optimum value for those programs. On the other hand, the matrix multiplication program extensively uses the multiplier more often than the other six benchmark programs. Because the fault detection probability in the array multiplier is the highest when k = -4, the fault detection probability in the matrix program is the highest when k = -4.

Finally, Table 10 shows the optimum value of k for each benchmark program.

6. Simulation Results

In Sec. 5, we determined the optimum value of k for each benchmark program by looking at the execution profile of the program. In this section, we will verify our decision on the optimum value of k by simulating the benchmark programs in a MIPS simulator. We built a MIPS simulator that reads an assembly program, executes instructions, and emulates the functional units at the gate level for a fault injection simulation. We injected a stuck fault into randomly selected node of the functional units and generated output patterns, which may be corrupted if the inputs provoke the fault. We generated two output patterns in each functional unit: one from the original program and the other from the transformed program. We compared their outputs, counted detected and undetected incorrect outputs, and averaged them over all simulated single stuck faults. They are weighted by execution profiles of the programs, and finally $C(k)$ and $D(k)$ are obtained. The results are shown in Table 11 and Table 12. Table 13 shows the optimum values of k determined by simulation for the benchmark programs. The reader can verify that Table 13 is identical to Table 10; this demonstrates that our results in Sec. 5 agrees with the simulation results.

The numbers in Table 11 and Table 12 are slightly different from the numbers in Table 8 and Table 9, in which we assume the uniform probability of all numbers in the entire range of the inputs. However, when we simulate benchmark programs in our MIPS simulator, we cannot assume the uniform probability of actual inputs to the programs. For example, the sorting program in the simulation does not sort the entire range of integer numbers (from -2^{31} to 2^{31}) available in a 32-bit machine. Instead, it sorts the numbers that are smaller than ten thousand. Thus, there are slight discrepancies between the expected probabilities and the values obtained from simulation.

Table 11. *D(k)* with various values of *k* simulated in benchmark programs. Note that Shaded areas indicate the highest data integrity in a row.

	k = -5	*k* = -4	*k* = -3	*k* = -2	*k* = -1	*k* = 2	*k* = 3	*k* = 4	*k* = 5
I-sort	1.0000	1.0000	1.0000	1.0000	0.9641	1.0000	0.9998	1.0000	0.9998
Q-sort	1.0000	1.0000	1.0000	1.0000	0.9585	1.0000	0.9998	1.0000	0.9998
Lzw	0.9900	0.9940	0.9882	0.9940	0.9940	0.9940	0.9852	0.9940	0.9785
Fib	1.0000	1.0000	1.0000	1.0000	0.9590	1.0000	1.0000	1.0000	1.0000
M-mul	1.0000	1.0000	1.0000	1.0000	0.9721	1.0000	0.9999	1.0000	0.9999
Shuffle	0.9876	0.9796	0.9872	0.9900	0.9680	0.9900	0.9830	0.9898	0.9763
Hanoi	1.0000	1.0000	1.0000	1.0000	0.8309	1.0000	0.9999	1.0000	0.9998

Table 12. *C(k)* with various values of *k* simulated in benchmark programs. Shaded areas indicate the highest fault detection probability under the condition that data integrity is the highest (shaded areas in Table 11).

	k = -5	*k* = -4	*k* = -3	*k* = -2	*k* = -1	*k* = 2	*k* = 3	*k* = 4	*k* = 5
I-sort	0.6762	0.6837	0.6768	0.6901	0.6683	0.4420	0.4540	0.4493	0.4459
Q-sort	0.6810	0.6761	0.6841	0.6862	0.6866	0.5173	0.4848	0.5231	0.4821
Lzw	0.7202	0.7158	0.7192	0.7312	0.7400	0.5147	0.4880	0.5324	0.4837
Fib	0.6974	0.7199	0.7027	0.7249	0.7551	0.5324	0.5035	0.5314	0.5011
M-mul	0.6847	0.6907	0.6848	0.6898	0.6411	0.5425	0.5167	0.5526	0.5093
Shuffle	0.5614	0.5604	0.5703	0.5678	0.6019	0.4895	0.4576	0.4982	0.4609
Hanoi	0.7403	0.7649	0.7407	0.7730	0.7769	0.5453	0.5172	0.5593	0.5135

Table 13. Optimum value of *k* determined for each benchmark programs by simulation.

	I-sort	Q-sort	Lzw	Fib	Mat-mul	Shuffle	Hanoi
Optimum *k*	-2	-2	-2	-2	-4	-2	-2

7. Floating Point Numbers

There are several ways that non-integers can be represented. Examples include using fixed point representation and using a pair of integers (*a*, *b*) to represent the fraction *a/b*. However, only *floating point* representation has gained widespread use [Hennessy 96]. In this floating point system, a computer word is divided into three parts: a sign, an exponent and a fraction. As an example, in the IEEE standard 754 [IEEE 85], single-precision numbers are stored in 32 bits: 1

bit for the *sign s*, 8 bits for the *exponent e*, and 23 bits for the *fraction f* as shown in Fig. 1. In this paper, we assume that floating point numbers are represented in the IEEE standard 754 format.

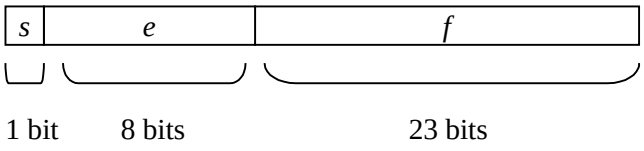


Fig. 7.1. A single-precision floating point number in IEEE standard 754.

The exponent is a signed number represented using the bias method with a bias of 127. The fraction *f* is the fraction part of the *mantissa*, i.e., *mantissa* = 1.*f*. Thus, the number being represented is $s \times 1.f \times 2^{e-127}$ [IEEE 85]. For example, a decimal number 1.5 can be represented as $1.1_2 \times 2^0$ in two's complement representation, because $1.5_{10} = 1 + 0.5 = 1_2 + .1_2 = 1.1_2 \times 2^0$ (*s* = 0, *f* = .1 and *e* = 127). Since floating point numbers and integers have different representations, floating point computation is usually distinguished from integer computation and performed in a special functional unit called a *Floating Point Unit* (FPU).

Having three parts in one word in floating point representation creates some difficulty in applying ED^{4}I to floating point numbers because multiplying by *k* may not shift nor change many bits in one word. For example, if the value of *k* is –1 and we multiply the original data by this *k*, only one bit of the word (the sign bit in the representation) is changed, and the rest of the word remains unchanged. If *k* is a power of 2, only some bits in the exponent portion *e* are changed, but the fraction part *f* is not changed. For example, consider a decimal number 1.5 again. In 2's complement representation, 1.5_{10} is $1.1_2 \times 2^{127}$ (*s* = 0, *e* = 127, *f* = .1). If we multiply 1.5_{10} by 2, the result is 3_{10} in decimal representation and $1.1_2 \times 2^{128}$ (*s* = 0, *e* = 128, *f* = .1) in 2's complement representation. As we can see, there is no change in fraction part *f* of 2's complement representation. In other words, the data in the transformed program are not much changed no matter whether we transform a program with the factor *k* or not; thus, we cannot guarantee data integrity if we have a stuck-at fault in the fraction part.

Our approach to solve this problem is to find a value of *k* for the fraction and the exponent separately and combine those values to get the best value for *k*.

7.1. The value of k for fraction

Firstly, let us consider the fraction part. We choose $k = \frac{3}{2}$ for the fraction part because it satisfies the following criteria: (1) guaranteed data integrity in the fraction, (2) no underflow in

the transformed program, (3) low probability of overflow, and (4) ability to easily locate an error. The detailed explanations are as follows.

1) Guaranteed data integrity

Let us denote a floating point number x as $s \times m \times 2^n$, where s is a sign, m is a mantissa, and n is an exponent. The mantissa m is always $1 \le m < 2$ because of normalization. Suppose stuck-at 1 fault occurs in the ith bit (from the most significant bit) of the mantissa m; then, it will add $e = 2^{-i}$ to m if it is provoked.

i) In the original program

If we denote x_e as a corrupted x by this stuck-at 1 fault,

(7) $\quad x_e = s \times (m+e) \times 2^n, 1 < m+e < 2.$

ii) In the transformed program

$$x' = kx = \tfrac{3}{2}x = s \times \tfrac{3}{2}m \times 2^n$$

Since $1 \le m < 2$, $\tfrac{3}{2} \le \tfrac{3}{2}m < 3$.

However, a mantissa cannot be greater than 2; thus, if $\tfrac{3}{2}m > 2$, the mantissa is right-shifted by 1 (divided by 2) and normalized. This normalization will add one to the exponent. In other words,

$$x' = \begin{cases} s \times (\tfrac{3}{2}m) \times 2^n, if\ \tfrac{3}{2}m < 2 \\ s \times (\tfrac{3}{4}m) \times 2^{n+1}, if\ 2 \le \tfrac{3}{2}m < 3 \end{cases}$$

The same stuck-at 1 fault corrupts the ith bit of the mantissa and adds $e = 2^{-i}$ as it did in the original program. If we denote x'_e as a corrupted x' by this stuck-at 1 fault,

(8) $$x'_e = \begin{cases} s \times (\tfrac{3}{2}m + e) \times 2^n, if\ \tfrac{3}{2}m < 2 \\ s \times (\tfrac{3}{4}m + e) \times 2^{n+1}, if\ 2 \le \tfrac{3}{2}m < 3 \end{cases}$$

iii) Comparison

We have to check if the value of x'_e is erroneously equal to the value of kx_e. If the values are same, data integrity is not guaranteed. From (7),

(9) $$kx_e = \tfrac{3}{2}x_e = s \times \tfrac{3}{2}(m+e) \times 2^n = \begin{cases} s \times \tfrac{3}{2}(m+e) \times 2^n, if\ \tfrac{3}{2}(m+e) < 2 \\ s \times \tfrac{3}{4}(m+e) \times 2^{n+1}, if\ 2 \le \tfrac{3}{2}(m+e) < 3 \end{cases}$$

The mantissas in (8) are $\tfrac{3}{2}m + e, \tfrac{3}{4}m + e$. The mantissas in (9) are $\tfrac{3}{2}m + \tfrac{3}{2}e, \tfrac{3}{4}m + \tfrac{3}{4}e$. When we compare the mantissas in (8) with those in (9), they are all different values. Thus, x'_e never equals

to kx_e, and the stuck-at 1 fault can be detected by comparing these two values. Therefore, data integrity is guaranteed.

Similarly, we can prove that data integrity is also guaranteed when stuck-at 0 fault occurs.

(2) No underflow

Since $k > 1$, we do not have an underflow in the floating point number of the transformed program.

(3) Low probability of overflow

When we multiply the data by $k < 2$, the exponent of the original data increases at most by one. If a maximum value of data is less than $2^{2^8-2-127} = 2^{127}$ in IEEE standard 754 format, we can guarantee no overflow in the transformed program.

(4) Ability to easily locate an error

If we subtract the mantissa of x'_e from the mantissa of kx_e, the result is either $\frac{1}{2}e$ or $\frac{1}{4}e$. Since e is the stuck-at fault in the ith bit, $\frac{1}{2}e$ is the i+1th bit (one bit right-shifted e) and $\frac{1}{4}e$ is the i+2th bit (two bit right shifted) in the mantissa. Therefore, when we want to locate an error, we perform $kx_e - x_e$; then, if the result is 2^{-j}, the stuck-at fault occurred in either j-1th bit or j-2th bit.

7.2. The value of k for exponent

Secondly, let us consider the value of k for the 8-bit exponent of the floating point number. We use two transformations to guarantee data integrity in the exponent. We choose $k = 2^{10101010_2}$ for the first transformation and $k = 2^{01010101_2}$ for the second transformation. In this case, we have assumed that these k's will not cause an overflow. (However, if there is an overflow, we can use the scaling techniques described in Sec. 4.)

When we multiply the original value by k, the exponent of k is added to the exponent of the original value. Thus, 10101010_2 is added to the exponent of the original data in the first transformation, and 01010101_2 is added to the exponent of the original data in the second transformation. Then, we will prove later that every bit of the exponent of the original value can be complemented either by the first transformation or by the second transformation. If every bit of the exponent field can be complemented in the transformed programs, a single stuck-at fault in the exponent field can be detected.

First, suppose a stuck-at fault is provoked and produced an error in the original program. Since every bit in the exponent can be complemented in the transformed programs, the stuck-at fault will not be provoked in at least one of the transformed programs. Thus, when we compare the corrupted value in the original program with the uncorrupted value in the transformed program, we can detect the error. Second, suppose the stuck-at fault is not provoked in the original program. Then, it will be provoked in at least one of the transformed programs and the data will be corrupted. We can detect the error by comparing the corrupted value with the uncorrupted value. Therefore, a single stuck-at fault in the exponent field can be detected.

Now, let us prove that every bit of the exponent of the original value can be complemented either by the first transformation or by the second transformation.

Proof. Let an exponent $n < 2^N$ can be represented by $n = \sum_{i=0}^{N-1} b_i 2^i$, where $b_i = 0$ or 1, and N is *the number of bits in the exponent*. For an integer $0 < h < N$, define n_h as $n_h = n \,\&\, (2^h - 1) = \sum_{i=0}^{h-1} b_i 2^i$, where & represents *logical AND* operation; then, n_h is always $0 \le n_h < 2^h$. First, let us assume h is an even number. Define p, p_h, q and q_h as:

$$p = \sum_{i=0}^{\frac{N}{2}-1} 2^{2i}$$

$$p_h = p \,\&\, (2^h - 1)$$

$$q = \sum_{i=0}^{\frac{N}{2}-1} 2^{2i+1}$$

$$q_h = q \,\&\, (2^h - 1)$$

Fig. 7.2 shows the numbers we have defined.

n	=	$b_{N-1}\ldots$	b_h	$b_{h-1}\ldots b_1 b_0$
n_h	=	0 0	0	$b_{h-1}\ldots b_1 b_0$
p	=	1 0 …	0	1 0 …1 0
p_h	=	0 0 …	0	1 0 …1 0
q	=	0 1 …	1	0 1 …0 1
q_h	=	0 0 …	0	0 1 …0 1

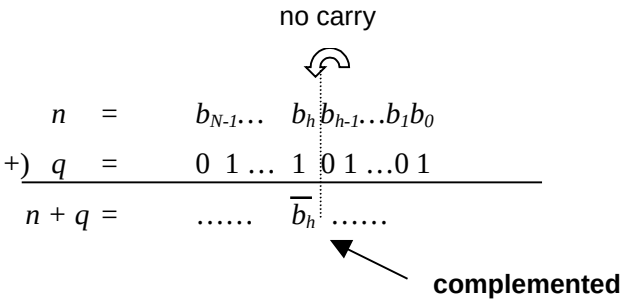
.

Fig. 7.2. n, n_h, p, p_h, q and q_h.

Now, let us define z_h as $z_h = 2^h - q_h < 2^h$; then, z_h represents the smallest number of n_h that can produce a carry from the bit b_{h-1} to b_h when it is added by q.

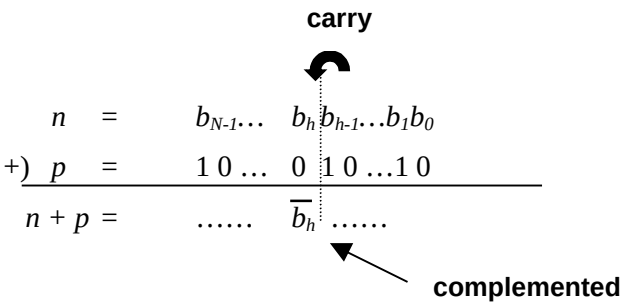
i) When $0 \le n_h < z_h$.

$n + q$ does not produce a carry from b_{h-1} to b_h because n_h is less than z_h. Therefore, b_h is complemented by adding q.



ii) When $z_h \le n_h < 2^h$.

Since $p_h > q_h$, $n_h + p_h > n_h + q_h > z_h + q_h = 2^h$; thus, $n + p$ will always produce a carry (2^h) from b_{h-1} to b_h. Therefore, b_h is complemented by adding p.



From i) and ii), b_h for any even number $0 < h < N$ is always complemented by either p or q. Similarly, we can prove that b_h is always complemented by either p or q when $0 \le h < N$ is an odd number or zero. Therefore, every bit of n is always complemented by either p or q.

7.3. The value of k for floating point numbers

We have selected $k = \frac{3}{2}$ for detecting an error in the fraction of a floating point number representation. We also have selected $k = 2^{10101010_2}$ and $k = 2^{01010101_2}$ for the exponent. As a result, we use $k_1 = -\frac{3}{2} \times 2^{10101010_2}$ for the first transformation and $k_2 = -\frac{3}{2} \times 2^{01010101_2}$ for the second transformation.

In a simulation experiment using $k_1 = -\frac{3}{2} \times 2^{10101010_2}$ and $k_2 = -\frac{3}{2} \times 2^{01010101_2}$, we analyzed the data integrity and the fault detection probability in the bus, CLA adder, and multiplier. We applied all possible binary input patterns to these functional units in the presence of a randomly selected fault and checked whether the fault was detected or not. We repeated simulations 10^4 times for each functional unit. Fig. 7.3 shows the simulation results. In Fig. 7.3, (a) shows that data integrity is guaranteed in these functional units. In Fig. 7.3 (b), we can see that data integrity is guaranteed in the bus, but not in the CLA adder. If the CLA adder has a stuck-at fault in some of carry signal lines, we miss less than 3% of the injected faults. Also, note that the multiplier is not shown in (b) because exponent multiplication is not required in floating point addition, subtraction, multiplication and division.

The simulation results show that using $k_1 = -\frac{3}{2} \times 2^{10101010_2}$ and $k_2 = -\frac{3}{2} \times 2^{01010101_2}$ can guarantee data integrity for all floating point computations except the exponent addition. Only in the exponent addition, data integrity is not guaranteed for 100%, but it is still higher than 97%.

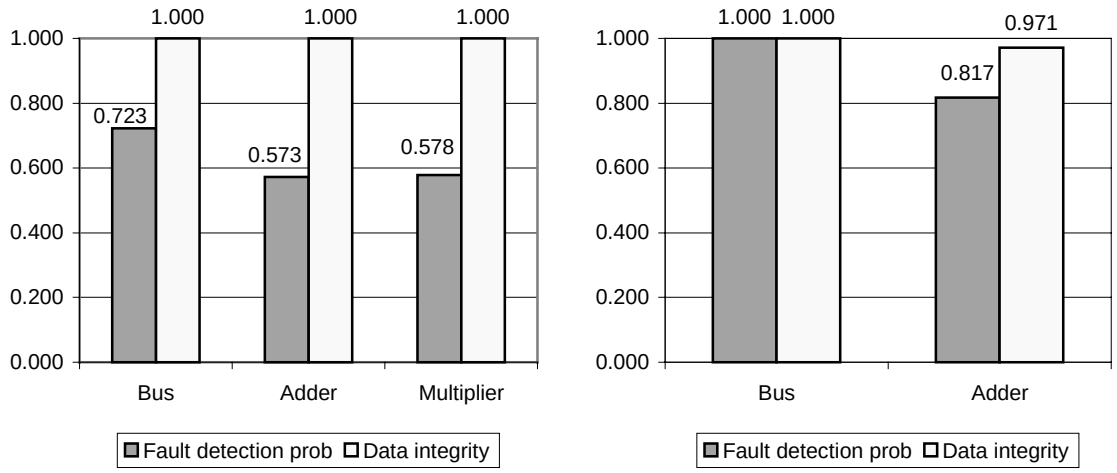


Fig. 7.3. Simulation results: Data integrity and fault detection probability in (a) fraction and (b) exponent of IEEE 754 single precision.

7.4. Using integer units for floating point computation

Our another approach to guarantee data integrity in floating point computation is to use an integer functional unit for floating point calculation in the transformed program. The original program uses the FPU whereas the transformed program uses the integer unit for the same

calculation. When we use integer units for floating point numbers, we can use available floating point number computation library or write our own code. It is true that using the integer unit for floating point calculation will cause performance overhead. However, this technique will guarantee data integrity because we use different hardware for the same computation. In this case, we use different resources for computation by converting a floating point number into three integers and performing separate integer computations.

For example, the pseudo code in Fig. 7.4 outlines major steps needed to perform multiplication. For simplicity, we omit the details such as adjusting the exponent bias, shifting the fraction and rounding numbers.

```
z = mul(x, y)
mul(float x, float y)
{
    fx = x AND MASK_FRACTION;
    fy = y AND MASK_FRACTION;
    fz = fx * fy;
    xe = x AND MASK_EXPONENT;
    ye = y AND MASK_EXPONENT;
    ez = ex + ey;
    sz = (x XOR y) AND MASK_SIGN;
    z = sz OR ez OR fz;
    return z
}
```

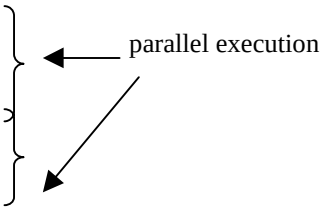


Fig. 7.4. A pseudo code using integer units for floating point multiplication.

In the code, MASK_FRACTION, MASK_EXPONENT, MASK_SIGN are constant mask bits. Performing logical AND with the mask bits, we extract a fraction and an exponent from the floating point number x and store them in f_x and e_x respectively. Similarly, f_y and e_y contain a fraction and an exponent extracted from y. Then, $f_x f_y$ produces f_z using an integer multiplier, and $e_x + e_y$ produces e_z using an integer adder. By combining s_z, e_z and f_z using logical OR, we can create a new result z.

Furthermore, execution overhead can be reduced by executing multiple instructions in parallel. As shown in the figure, exponent addition and fraction multiplication can be executed simultaneously in super-scalar and VLIW processors. For example, an integer multiplication takes 5 cycles and an integer addition takes 1 cycle in MIPS R10000, and they can be executed in parallel. While computing multiplication $f_x f_y$, we can concurrently calculate exponent using the adder instead of idling the adder. Timing diagram of this parallel execution is shown in Fig. 7.5. The timing diagram shows only major steps described in Fig. 7.4, but readers can clearly see that parallel execution reduces execution time overhead. For example, in MIPS R10000, floating

point multiplication using FPU takes 2 cycles. In this case, execution overhead will be 7/2 = 3.5 (assuming no rounding of numbers). In contrast, serial execution of the pseudo code will take 10 cycles (3 cycles for sign, exponent and fraction extraction, 1 cycle for addition, 5 cycles for multiplication, 1 cycle for XOR, and 1 cycle for OR), resulting in overhead of 10/2=5. We can achieve 30% speed-up by parallel execution.

Floating point addition, subtraction and division algorithms are described in detail in [Hennessy 96] and can be also implemented in software using integer unit. There is a possibility that we will have precision errors between the computation results from integer unit and those from FPU. In this case, we need to check whether the difference between the two results is within a threshold or not, rather than check exact match between them.

In summary, using different hardware – integer unit and floating point unit – for the same computation will guarantee data integrity for the system. However, floating point computations using integer unit will cause performance overhead.

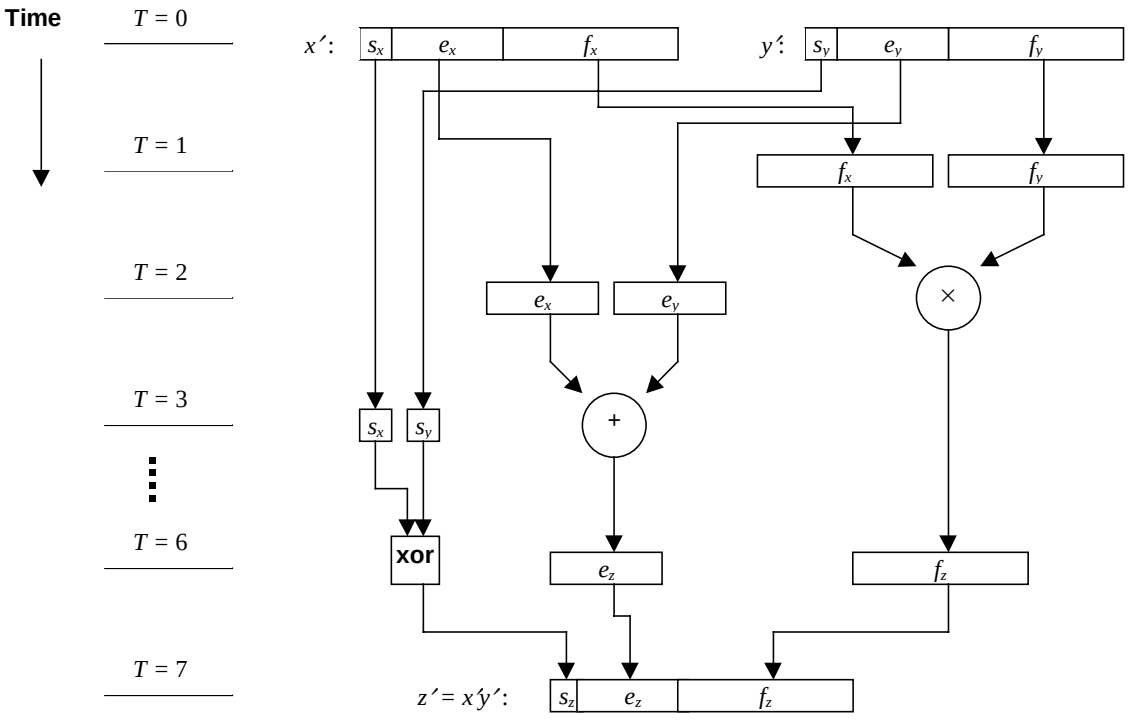


Fig. 7.5. Timing diagram for parallel execution of the code in Fig. 7.5

8. Summary

In this paper, we have presented the ED^{4}I technique based on data diversity to detect hardware faults in the system without any hardware modification. Unlike previous techniques, our technique is a pure software method that can be easily implemented in any system. ED^{4}I transforms a program to a new program with the same functionality but with diverse data without changing the complexity of the original program.

The factor *k* determines how diverse the transformed program is. To determine the optimum value of *k*, we have used data integrity and fault detection probability as metrics to quantify the diversity of the transformed program. Based on these metrics, we demonstrated how to choose the optimum value of *k* for creating data diversity. For example, we have observed that –2 is the optimum value for *k* in six out of seven benchmark programs we have simulated.

The execution profiles of those six programs show that adders are the most frequently used functional units in the programs; thus, the transformation with $k = -2$ might be the most desirable choice for the programs that use adders extensively. However, program profiling and fault injection simulations in functional units are necessary to get the exact optimum value for *k* because data integrity and fault detection probability depends on the implementation of functional units.

If a hardware design is fixed (such as in COTS) and cannot be modified, we can use ED^{4}I to improve the data integrity of the system. Transforming the original program of the system, running the original and transformed program, and subsequently comparing outputs can improve the data integrity and availability of the system.

9. Acknowledgement

This work was supported in part by the Ballistic Missile Defense Organization, Innovative Science and Technology (BMDO/IST) Directorate and administered through the Department of the Navy, Office of Naval Research under Grant Nos. N00014-92-J-1782 and N00014-95-1-1047..

10. References

[Avizienis 77] Avizienis, A. and L. Chen, "On the implementation of N-version programming for software fault-tolerance during program execution," *Proc. Intl. Computer software and Appl. Conf.*, pp. 145-155, 1977

[Avizienis 84] Avizienis, A. and J.P.J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," *IEEE Computer*, pp. 67-80, Aug. 1984.

[Brown 60] Brown, D. T., "Error detecting and correcting binary codes for arithmetic operations," *IRE Trans. on Electronic Computers*, vol. EC-9, pp. 333-337, Sep. 1960

[Chakravarty 90] Chakravarty, S. and H. B. Hunt, "On Computing Signal Probability and Detection Probability of Stuck-at Faults," *IEEE Trans. on Computers*, vol. 39, no. 11, pp. 1369-77, Nov. 1990.

[Chen 78] Chen, L., and A. Avizienis, "N version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," *Proceedings of the 8th Intl. Symposium on Fault-Tolerant Computing*, pp. 3-9, Toulouse, France, 1978.

[David 90] David, R. and K. Wagner, "Analysis of Detection Probability and Some Applications," *IEEE Trans. on Computers*, vol. 39, no. 10, 1284-91, Oct. 1990.

[Dugan 93] Dugan, J. B., and R. V. Buren, "Reliability Evaluation of Fly-by-Wire Computer Systems," *Journal of Systems and Software*, vol. 25, no. 1, pp. 109-20, Apr. 1994.

[Eckhardt 85] Eckhardt, D.E., and Lee, L.D., "A Theoretical Basis for the Analysis of Multi-version Software Subject to Coincident Errors," *IEEE Trans. on Software Engineering*, vol. SE-11, no. 12, pp. 1511-17, 1985.

[Harrison 77] Harrison, W. "Compiler Analysis of the Value Ranges for Variables," *IEEE Trans. on Software Engineering*, vol. SE-3, no. 3, pp. 243-50, May 1977

[Hennessy 96] Hennessy, J. L. and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Second edition, 1996.

[Kapur 92] Kapur, R. and M. R. Mercer, "Bounding Signal Probabilities for Testability Measurement Using Conditional Syndromes," *IEEE Trans. on Computers*, vol. 41, no. 12, pp. 1580-88, Dec. 1992

[Lala 94] Lala, J. H. and R. E. Harper, "Architectural principles for safety-critical real-time applications," *Proc. of the IEEE*, vol. 82, no. 1, pp. 25-40, Jan. 1994.

[Laprie 90] Laprie, J. C., and et al, "Definition and Analysis of Hardware and Software Fault Tolerant Architectures," *IEEE Computer*, vol. 23, no. 7, pp. 39-51, July 1990.

[Litt 89] Littlewood, B., and Miller, D.R., "Conceptual Modeling of Coincident Failures in Multiversion Software," *IEEE Trans. on Software Engineering*, vol. 15, no. 12, pp. 1596-1614, 1989

[Lyu 91] Lyu, M. R. and A. Avizienis, "Assuring design diversity in N-version software: a design paradigm for N-version programming," *Proc. DCCA*, pp. 197-218, 1991

[Mitra 99] Mitra, S., N. Saxena and E. J. McCluskey, "A design diversity metric and reliability analysis for redundant systems," *Intl. Test Conference*, pp. XX 1999.

[Parker 97a] Parker, K. and E. J. McCluskey, "Analysis of Logic Circuits with Faults Using Input Signal Probabilities," *IEEE Trans. on Computers*, vol.C-54, no. 5, pp. 573-8, May 1975.

[Parker 97b] Parker, K. and E. J. McCluskey, "Probabilistic Treatment of General Combinational Networks," *IEEE Trans. on Computers*, vol. C-24, no. 6, pp. 668-70, June 1975.

[Patel 82] Patel, J. H. and L. Y. Fung, "Concurrent Error Detection in ALU's by Recomputing with Shifted Operands," *IEEE Trans. on Computers*, vol. C-31, no. 7, pp. 589-95, July 1982.

[Patel 83] Patel, J. H. and L. Y. Fung, "Concurrent Error Detection in Multiply and Divide Arrays," *IEEE Trans. on Computers*, vol C-32, no. 4, pp. 417-22, April 1983.

[Patterson 95] Patterson, J. "Accurate Static Branch Prediction by Value Range Propagation," *Proceedings of the SIG-PLAN Conference on Programming language Design and Implementation*, vol. 37, pp. 67-78, June 1995.

[Scot 83] Scott, R. K., J. W. Gault, and D. F. McAllister, "The Consensus Recovery Block," *Proceedings of the Total systems Reliability Symposium*, pp. 74-85, Dec. 1983.

[Scot 87] Scott, R. K., J. W. Gault, and D. F. McAllister, "Fault-Tolerant Reliability Modeling," *IEEE Trans. on Software Engineering*, vol. SE-13, no. 5, pp. 582-92, 1987.

[Shedletsky 78] Shedletsky, J. J., "Error Correction by Alternate-Data Retry," *IEEE Trans. on Computers*, vol. C-27, no. 2, pp. 106-12, Feb. 1978.

[Stephenson 00] Stephenson, M., J. Babb and S. Amarasinghe, *Proceedings of the SIGPLAN conference on Programming Language Design and Implementation*, Vancouver, British Columbia, June 2000.

[Sterbenz 74] Sterbenz, Pat H., *Floating-point computation,* Englewood Cliffs, N.J., Prentice-Hall, 1974.

[Weste 92] Neil H. Weste and K. Eshraghian, *Principles of CMOS VLSI Design,* Addison Wesley, 1992.

Appendix I: Program Transformation Algorithm

An expression $expr(x_1, x_2,\ldots,x_n)$ represents arithmetic or logical operations on variables $x_1, x_2,\ldots,x_n$. An *assignment* statement $y := expr(x_1, x_2,\ldots,x_n)$ defines y by assigning y the result of $expr(x_1, x_2,\ldots,x_n)$. A *branch* statement *if* $expr(x_1, x_2,\ldots,x_n)$ determines the control flow according to the result of $expr(x_1, x_2,\ldots,x_n)$. A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without branching except at the end.

By defining $V = \{v_1, v_2,\ldots,v_n\}$ as the set of vertices denoting basic blocks, and $E = \{(i,j) \mid (i, j)$ is a branch from v_i to $v_j\}$ as the set of edges denoting possible flow of control between the basic blocks, a program can be represented by a *program graph*, $P_G = \{V, E\}$.

The program transformation consists of two parts: expression transformations in basic blocks and branching condition transformations in branch statements. The expression transformation converts all the variables and constants in P' to k-multiples of corresponding variables and constants in P. Since the values of the variables and constants are different in P and P', the branching condition transformation modifies the expression in the branch statement and keeps the control flow in P' as same as the one in P.

Expression Transformation

Let $x'_1, x'_2,\ldots, x'_n$ be the variables in the transformed program P' and correspond to the variables $x_1, x_2,\ldots, x_n$ in P; then, $k \cdot x_i = x'_i$, $i = 1,2,\ldots,n$ should be always true during run time. Furthermore, any expression $expr'(x'_1, x'_2,\ldots, x'_n)$ in P' corresponding to $expr(x_1, x_2,\ldots, x_n)$ in P also should satisfy $k \cdot expr(x_1, x_2,\ldots, x_n) = expr'(x'_1, x'_2,\ldots, x'_n)$, which is achieved by the expression transformation to be described in this section.

An expression $expr(x_1, x_2,\ldots, x_n)$ can be recursively represented by a *parse tree*, a tree in which the leaves are the variables $x_1, x_2,\ldots, x_n$ or constants. An interior node in the parse tree has three children: the child on the left and right is either a node or a leaf, and the child in the middle denotes an arithmetic operation. An example is shown in Fig. I.1.

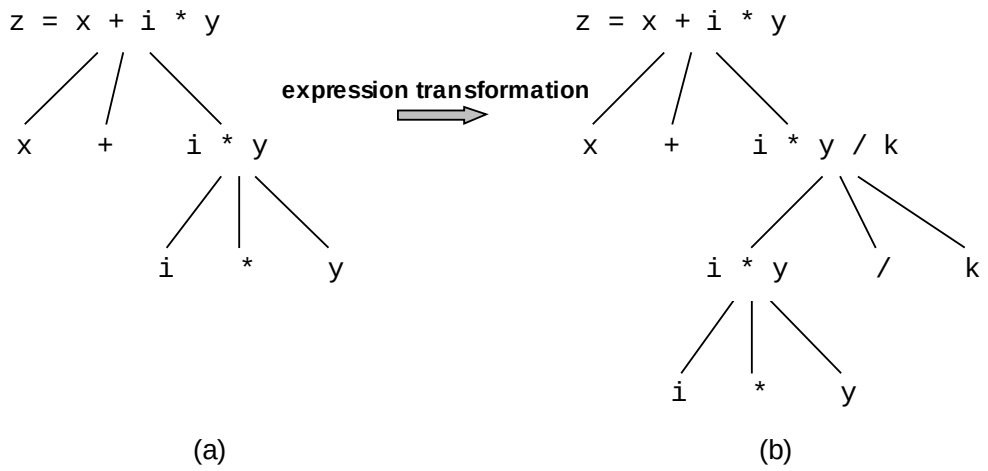


Fig. I.1 An example of a parse tree of `z = x + i * y` (a) before transformation (b) after transformation.

A parse tree *T* is built from an expression $expr(x_1, x_2,..., x_n)$ in *P*. Suppose *T'* represents a parse tree for a corresponding new expression $expr'(x'_1, x'_2,...x'_n)$ in *P'*; then, the values represented by the internal nodes in *T'* should be *k*-multiple of the values in the corresponding nodes in *T*.

An algorithm recursively transforming $expr(x_1, x_2,...,x_n)$ to $expr'(x'_1, x'_2,\ldots, x'_n)$ follows:

```
1 transform (t) {
2     if (leaf)
3           if (constant) return x' = k * x;
4           else  return x'
5     else {
6           left_child(t') := transform (left_child(t));
7           right_child(t') := transform (right_child(t));
8           middle_child(t') := middle_child(t);
9           if (middle_child(t) = * (or /)) {
10                right_child(t'') := k;
11                left_child(t'') = t';
12                middle_child(t'') = / (or *);
13                return t'';
14          } else {
15                return t';
16          }
17    }
18 }
```

Theorem I.1.. The internal nodes in T' = `transform(`T`)` are always *k*-multiple of the corresponding nodes in *T*.

Proof. By induction.

Base case of the induction: All the leaves in T' are k-multiple of the leaves of T by line 2,3 and 4 of the algorithm.

Induction step: Let us represent n as the result of the operation with left and right child of one internal node of T, and n' as the result of the corresponding node of T'. Assume that left and right child of the internal node in T' are k-multiple of the left and right child of the corresponding node in T. If the operation is an addition, n' is still k-multiple of n, i.e., $k \cdot n = n'$. If the operation is multiplication, n' is $n' = k \cdot$left_leftchild $\cdot\ k \cdot$ right_child $= k \cdot k \cdot n$. Line 9 to 12 eliminates this extra k and keeps $n' = k \cdot n$. Similarly, it is also true that $n' = k \cdot n$ in division. Therefore, `transform`(t) always returns t' such that the value of the top node of t' is always k-multiple of the top node of t.

Branching Condition Transformation

The branch statement compares two values (or expressions) and determines the control flow based on the comparison result. The expression in the branch statement has binary values: true or false. For example, if the expression in the branch statement is true, the branch is taken. If it is false, the branch is not taken. The branch statement can be represented by a decision triangle as shown in Fig. I.2. If $expr(x_1, x_2,...,x_n)$ is true, the branch is taken, and otherwise, the next statement is executed. This control flow should be preserved in $expr'(x'_1, x'_2,\ldots, x'_n)$ in P', i.e., when $k \cdot x_i = x'_i$, $expr'(x'_1, x'_2,\ldots, x'_n)$ always takes the same value (true or false) as $expr(x_1, x_2,...,x_n)$ in P so that the control flow in P' always identical to the one in the original program.

An algorithm for branching condition transformation is:

1. $expr(x_1, x_2,...,x_n)$ in a branch statement is transformed to $expr'(x'_1, x'_2,\ldots, x'_n)$ by expression transformation
2. If $expr(x_1, x_2,...,x_n)$ contains $\geq$, $>$, $\leq$, or $<$
3. if $k < 0$, then $\geq$, $>$, $\leq$, or $<$ is converted to $\leq$, $<$, $\geq$, or $>$ respectively.

Theorem I.2. The branching condition transformation makes the two control flows determined by $expr(x_1, x_2,...,x_n)$ and $expr'(x'_1, x'_2,\ldots, x'_n)$ always identical.

Proof. Inequality relationship such as $x_1 < x_2$ is preserved when both sides are multiplied by positive k. If multiplied by negative k, the inequality is reversed as in $x_1 > x_2$ by line 2 and 3.

Therefore, if $k \cdot x_i = x'_i$, $expr(x_1, x_2,...,x_n)$ and $expr'(x'_1, x'_2,..., x'_n)$ have the same true or false value, and the control flows determined by $expr(x_1, x_2,...,x_n)$ and $expr'(x'_1, x'_2,..., x'_n)$ are the same.

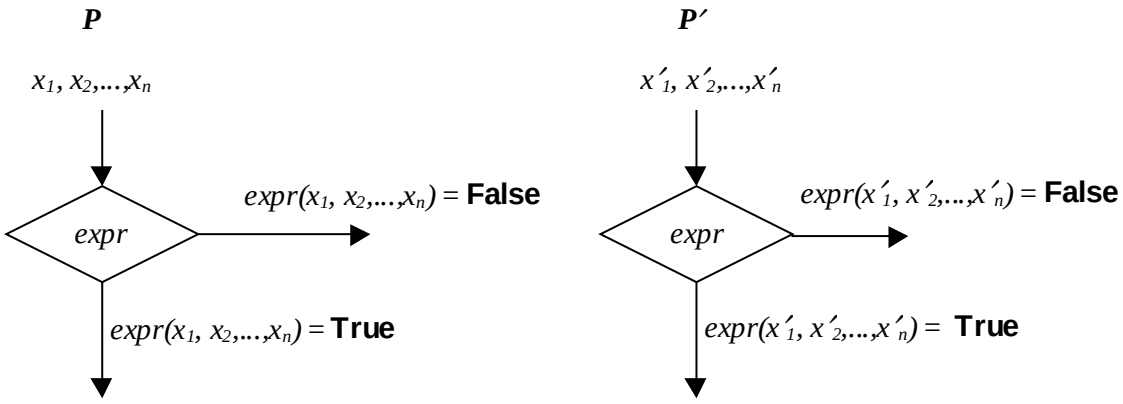


Fig.I.2. Branch determined by the expression in a branch statement.

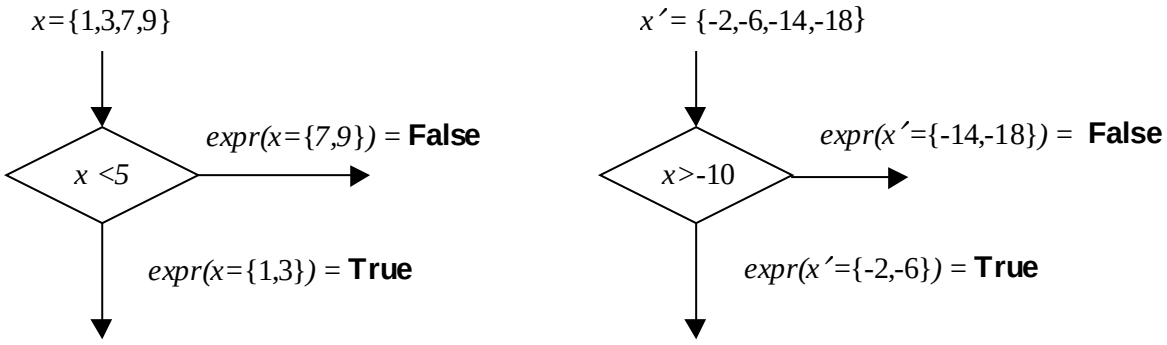


Fig.I.3. Branching in the original and transformed expressions when $k=-2$

Figure I.3 illustrates an example control flow when $k \cdot x_i = x'_i$, and k = -2. The branch statement expression $expr(x)=\{x < 5\}$ in (a) is transformed to $expr'(x')=\{x' > -10\}$ in (b). When x = {7, 9}, the branches are taken. When x' = {-14, -18} that is k-multiple of x = {7, 9}, k = -2, the branches are also taken in the transformed branch statement. Similarly, the branches are not taken both in x = {1, 3} and x' = {-2, -6}, and the two control flows by the original and transformed expressions are identical.

Program Transformation

If $k \cdot S(n) = S'(n)$ after n vertices are executed, *Theorem I.1* tells us that $k \cdot S(n+1) = S'(n+1)$ after one more vertex is executed. Also consider a branch statement s in P and the corresponding branch statement s' in P'. If $k \cdot S(n) = S'(n)$, the control flows determined by s and s' are identical by *Theorem I.2*.

(I.1) If $k \cdot S(n) = S'(n)$, then $k \cdot S(n+1) = S'(n+1)$

(I.2) For $\forall v_i \in V$ that have a branch statement s_i at the end and are executed after m-1 vertices are executed:

if $k \cdot S(m) = S'(m)$ and, s_i and s'_i are executed, then branching determined by s' is always identical to branching by s.

Let us denote the first basic block in P and P' by v_0 and v'_0, and assume that $S(0) = S'(0)$ when the program starts. This constitutes the base case of the induction, and the inductive step (I.1) and (I.2) inductively proves that the control flows in P and P' are always identical. This proof leads us to *Theorem I.3*, which shows us that during run time, the variables in the transformed program are always kept as k-multiple of the corresponding variables of the original program.

Theorem I.3. For $n>0$, it is always true that $k \cdot S(n) = S'(n)$.

Appendix II

This section discusses how to get a closed form solution for an optimal value of k that maximizes fault detection probability under the condition that data integrity is maximized. We discuss two functional units as examples: a bus and an adder. Sec. II.1 considers a bus signal line that consists of multiple parallel signal lines. Sec. II.2 considers a ripple carry adder as an example of iterative logic array.

II.1 Bus Signal Line

An M bit bus consists of M parallel signal paths. As in Fig. 5.1, the source places an M bit signal x on the M bit bus and transfers information to the destination. If the ith bit of the bus has a stuck fault, the destination may receive a corrupted value of x.

We can detect this fault by placing x of the original program and x' (= kx) of the transformed program on the bus one after the other and comparing the received values [Shedletsky 78]. If the fault corrupts either x and x' in different way at the destination after x and x' are received , $x' = k \cdot x$ will not be satisfied. If x or x' does not provoke the fault, the destination receives the correct values and $x' = k \cdot x$ is satisfied, i.e., the fault does not corrupt the information.

1) k = -1

In 2's complement representation, negating a number is equivalent to reversing all the bits until the first 1 as shown in Fig. II.1 (a). Suppose that $\mathbf{b_i}$ is the first 1 in x. When x and x' (= kx = $-x$) are applied on the bus, all the bits from $\mathbf{b_{i+1}}$ to $\mathbf{b_{M-1}}$ in x' are complements of the values of $\mathbf{b_{i+1}}$ to $\mathbf{b_{M-1}}$ in x. Because each bit from $\mathbf{b_{i+1}}$ to $\mathbf{b_{M-1}}$ can have the value of 0 and 1 by x and x', any stuck fault in those signal lines are provoked, and the fault can be detected by comparing two values.

Theorem II.1. Assuming that input x is randomly chosen with equal probability, if a stuck-at fault exists on the bit $\mathbf{b_i}$ of the bus, the data integrity of x and x' for the bus is 1 when k = -1 and

$$\Pr\{k \cdot x \neq x'\} = 1 - \frac{1}{2^{i+1}} .$$

Proof. Figure II.1 (b) helps us to calculate $\Pr\{kx \neq x'\}$ when a stuck fault is present on the ith bit of the bus. Assume we have an Mbit bus. Since the MSB bit is a sign bit, 2^{M-1} positive numbers can be represented. If x is positive, x' is always negative. If x is negative, x' is always positive. Thus, we need to consider only positive x to calculate $\Pr\{kx \neq x'\}$. These numbers are grouped according to the position of the first 1 from $\mathbf{b_0}$ as shown in Fig. II.1 (b). In Fig. II.1, x represents the bit whose value is complemented when the number is negated. First, suppose $\mathbf{b_i}$ is stuck-at 0 (We also denote stuck-at 0 as s/0). The numbers from the first row to the i+1th row can provoke the fault because either x or x' is always 1 as shown in Fig. II.1 (b). Therefore, by adding the numbers shown in the entries from the first row to the i+1th row of the last column in Fig.II.1, we can calculate $\Pr\{kx \neq x'\}$ as:

$$\Pr\{k \cdot x \neq x' \mid \text{s/0 at } \mathbf{b_i}\} = \frac{1}{2^{M-1}}\left(2^{M-2} + 2^{M-3} + \ldots + 2^{M-i-2}\right) = 1 - \frac{1}{2^{i+1}} .$$

Second, suppose the ith bit is s/1; then, the numbers except for the i+1th row can provoke the fault.

$$\Pr\{\boldsymbol{k} \cdot \boldsymbol{x} \neq \boldsymbol{x}' \mid \text{s/1 at } \mathbf{b_i}\} = \frac{1}{2^{\boldsymbol{M}-1}}\left(2^{\boldsymbol{M}-1} - 2^{\boldsymbol{M}-\boldsymbol{i}-2}\right) = 1 - \frac{1}{2^{\boldsymbol{i}+1}} .$$

Therefore,

$$\Pr\{k \cdot x \neq x'\} = \frac{1}{2}\{\Pr\{k \cdot x \neq x' \mid \text{s/0 at } \mathbf{b_i}\} + \Pr\{k \cdot x \neq x' \mid \text{s/1 at } \mathbf{b_i}\}\} = 1 - \frac{1}{2^{i+1}}.$$

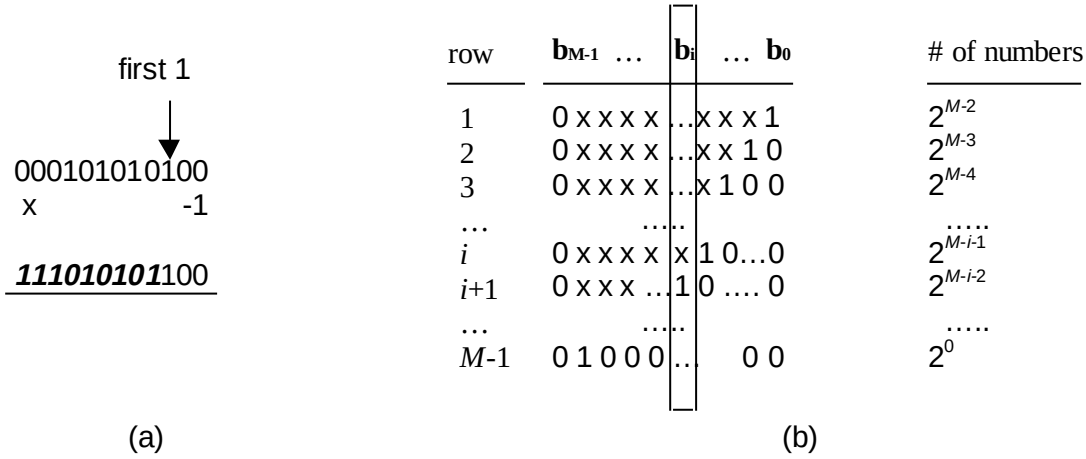


Fig.II.1 Negating numbers complements the bits until the first 1 from LSB $\mathbf{b_0}$

2) k = 2

Multiplying by 2 is equivalent to shifting one bit to the left (assuming the most significant bit MSB is on the left). Suppose we have an M bit bus and the bit i has stuck-at-0 fault. If a variable x is one of the variables in the original program and x' is a corresponding variable in the diverse program, x' is one bit left shifted x. When x is applied to this bus, the ith bit of x is corrupted if it is 1, and the corrupted value is $x - 2^i$. If x' is applied to the bus and the ith bit is corrupted, it could be seen as the same case as the i-1th bit of x is corrupted and left shifted; so, the corrupted value is $x' = 2(x - 2^{i-1}) = 2x - 2^i = kx - 2^i \neq kx$. Hence, if at least one of x and x' is corrupted, $k \cdot x = x'$ is not satisfied, so the comparison of two values will detect the fault. If neither of them is corrupted – both the (i-1)th and ith bit of x are zero – the fault (stuck-at-0) does not affect the output, and $k \cdot x = x'$ is satisfied.

Theorem II.2. Assume that input x is randomly chosen with uniform probability and no overflow in the bus. If k = 2 and a stuck-at fault exists in a bus, we can detect the fault with the probability of 0.75 and the data integrity for the bus is 1.

Proof. Suppose the ith bit of an M bit bus is stuck-at-0. Consider the i-1 and ith bit of x that is applied to this bus. The s/0 fault is provoked if the ith bit of x is 1. The i-1th bit of x is equivalent to the ith bit of x' if x is shifted by one bit left; thus, if the i-1th bit of x is 1, the s/0

fault is provoked. Four combinations are possible for the two bits: 00, 01, 10, and 11 as shown in Figure II.2. Three of them have at least one 1 and they can provoke the fault so that $k \cdot x = x'$ is not satisfied. Similarly, s/1 fault is also provoked by the three of four possible combinations; therefore, if a stuck fault is present on the bus, $\Pr\{k \cdot x \neq x'\} = 0.75$, and with this probability, we can detect the fault by mismatch.

If the fault is not provoked, it does not change x or x'; thus, $k \cdot x = x'$ is satisfied. Therefore, the data integrity of x and x' for the bus is:

$1 - \Pr\{ k \cdot x = x',$ and both of x and x' are corrupted in the same way$\} = 1 - 0 = 1$

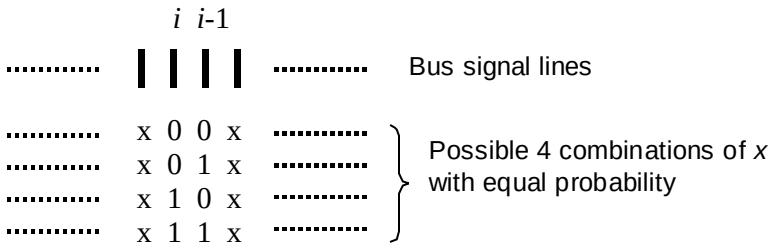


Fig.II.2 A bus signal lines applied by x

3) k = -2

Multiplying -2 is equivalent to negating a number and shift one bit to the left. Suppose $\mathbf{b_j}$ is the first 1 in x. All the bits from $\mathbf{b_{j+1}}$ to $\mathbf{b_{M-1}}$ in x are complemented and left shifted by one bit when x is multiplied by -2.. Each bit from $\mathbf{b_{j+1}}$ to $\mathbf{b_{M-1}}$ has the same probability of 0.5 for 0 and 1, and the probability remains the same although the bit is complemented. If $\mathbf{b_i}$, $j < i \leq \boldsymbol{M}$-1, is s/0 in the bus, 01, 10, and 11 in $\mathbf{b_i b_{i-1}}$ will provoke the fault, and the probability of detecting the fault is 0.75. Similarly, the s/1 in $\mathbf{b_i}$ is provoked with the probability of 0.75.

Theorem II.3. Assuming that input x is randomly chosen with equal probability, if a stuck-at fault exists in $\mathbf{b_i}$, the data integrity for the bus is 1 when k = -2 and

$$\Pr\{k \cdot x \neq x'\} = \frac{3}{4} - \frac{1}{2^M}$$

Proof. Suppose $\mathbf{b_i}$ is s/0 as shown in Fig. II.1. The numbers from the first row to the *i*-1th row can provoke the fault with the probability of 0.75 since the values of the bits denoted by x are complemented and shifted when multiplied by –2. The number of those numbers is $\left(2^{M-2}+2^{M-3}+...+2^{M-i}\right)$. The numbers represented by the *i*th and *i*+1th rows can provoke the fault with probability 1 since $\mathbf{b_{i-1}}$ and $\mathbf{b_i}$ are always 1, respectively. The number of those numbers is $2^{M-i-1}+2^{M-i-2}$. Therefore, we can calculate $\Pr\{kx \neq x' \mid \text{s/0 at } \mathbf{b_i}\}$ as:

$$\begin{aligned}\Pr\{k\cdot x \neq x' \mid \text{s/0 at } \mathbf{b_i}\} &= \frac{1}{2^{M-1}}\left\{\frac{3}{4}\left(2^{M-2}+2^{M-3}+...+2^{M-i}\right)+2^{M-i-1}+2^{M-i-2}\right\}\\ &= \frac{1}{2^{M-1}}\left\{\frac{3}{4}2^{M-i}\left(2^{i-1}-1\right)+2^{M-i-1}\left(1+\frac{1}{2}\right)\right\}=\frac{1}{2^{M-1}}\left\{\frac{3}{4}2^{M-1}\right\}\\ &= \frac{3}{4}.\end{aligned}$$

Now, suppose $\mathbf{b_i}$ is s/1. The numbers from the first row to the *i*-1th row can provoke the fault with the probability of 0.75. The number of those numbers is $2^{M-2}+2^{M-3}+...+2^{M-i}$. A half of the numbers in the *i*th row can provoke the fault because $\mathbf{b_{i-1}}$ is always 1 and $\mathbf{b_i}$ can be either 0 or 1 with equal probability. The number of these numbers is 2^{M-i-1}. The *i*+1th row to the last row can always provoke the fault because $\mathbf{b_{i-1}}$ or $\mathbf{b_i}$ is always 0. The number of these numbers is $2^{M-i-2}+...+2^0$. Thus, we can calculate $\Pr\{k\cdot x \neq x' \mid \text{s/1 at } \mathbf{b_i}\}$ as:

$$\begin{aligned}\Pr\{k\cdot x \neq x' \mid \text{s/1 at } \mathbf{b_i}\} &= \frac{1}{2^{M-1}}\left\{\frac{3}{4}\left(2^{M-2}+2^{M-3}+...+2^{M-i}\right)+2^{M-i-1}\cdot\frac{1}{2}+\left(2^{M-i-2}+...+2^0\right)\right\}\\ &= \frac{1}{2^{M-1}}\left\{\frac{3}{4}2^{M-i}\left(2^{i-1}-1\right)+2^{M-i-1}\frac{3}{2}-1\right\}=\frac{1}{2^{M-1}}\left\{\frac{3}{4}2^{M-1}-1\right\}\\ &= \frac{3}{4}-\frac{1}{2^{M-1}}.\end{aligned}$$

Therefore, if *M* is large enough, the probability approximates to:

$$\begin{aligned}\Pr\{k\cdot x \neq x'\} &= \frac{1}{2}\{\Pr\{k\cdot x \neq x' \mid \text{s/0 at } \mathbf{b_i}\}+\Pr\{k\cdot x \neq x' \mid \text{s/1 at } \mathbf{b_i}\}\}\\ &= \frac{1}{2}\{\frac{3}{4}+\frac{3}{4}-\frac{1}{2^{M-1}}\}=\frac{3}{4}-\frac{1}{2^M}\approx\frac{3}{4}\end{aligned}$$

II.2. Cell Array- Ripple Carry Adder

Let us denote signal names by bold letters and corresponding transition probabilities by italic letters. Table 14 summarizes several lemmas presented in [Parker 75a], which relates Boolean operations to corresponding operations on probabilities. Their proofs can be found in [Parker 75b].

Table 14. Summary of probabilistic model lemmas

Function	Probability	Assumption
a	a	--
a′	$1 - a$	--
ab	ab	**a, b** independent
$\mathbf{x_1 x_2 \ldots x_n}$	$x_1 x_2 \ldots x_n$	All $\mathbf{x_i}$ independent
$\mathbf{a} \vee \mathbf{b}$	$a + b - ab$	**a, b** independent
$\mathbf{x1} \vee \mathbf{x2} \vee \ldots \vee \mathbf{x_n}$	$1 - \prod_{i=1}^{n} (1 - x_i)$	All $\mathbf{x_i}$ independent
$\mathbf{a} \vee \mathbf{b}$	$a + b$	**a,b** not simultaneously 1
$\bigcup_{i=1}^{n} \mathbf{x_i}$	$\sum_{i=1} x_i$	$x_i \cdot x_j = 0 \ \ \forall\ i \neq j$

In this section, we will show how to get a closed form solution of the fault detection probability when we use $k = 2$ for a ripple carry adder.

Let us consider Fig.II.3 that shows one cell C_i (a full adder) of a ripple carry adder. In Fig.II.3, a_i, b_i, c_i and c_{i+1} denote the probability that $\mathbf{a_i}$, $\mathbf{b_i}$, $\mathbf{c_i}$ and $\mathbf{c_{i+1}}$ become 1. We can derive c_{i+1} from a_i, b_i and c_i using lemmas in Table 14:

$$\begin{aligned} c_{i+1} &= 1 - (1 - a_i b_i)(1 - c_i a_i)(1 - c_i b_i) \\ &= a_i b_i + c_i a_i + c_i b_i - a_i^{\,2} b_i c_i - a_i b_i^{\,2} c_i - a_i b_i^{\,2} c_i + a_i^{\,2} b_i^{\,2} c_i^{\,2} \end{aligned}$$

By using *Lemma5* in [Parker 75b], we can suppress the exponents in the expression. Therefore,

$$\begin{aligned} c_{i+1} &= a_i b_i + c_i a_i + c_i b_i - 3 a_i b_i c_i + a_i b_i c_i \\ &= a_i b_i + c_i (a_i + b_i) - 2 a_i b_i c_i \end{aligned}$$

Assuming $a_i = b_i = 0.5$, we can solve the equation and obtain a closed form for $\mathbf{c_i}$:

$$c_{i+1} = \frac{1}{4} + \frac{1}{2} c_i$$

$$c_i = \frac{1}{2} - \frac{1}{2^{i+1}}, i \geq 1, \ c_0 = 0$$

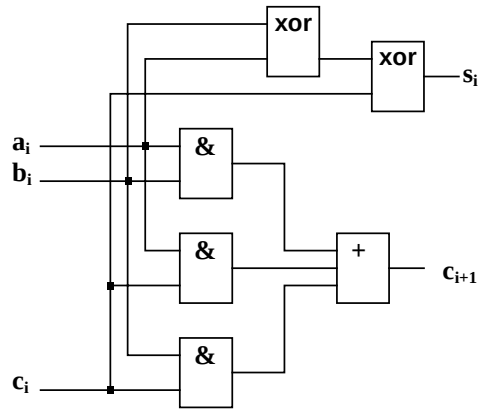


Fig.II.3 One cell C_i of a full adder

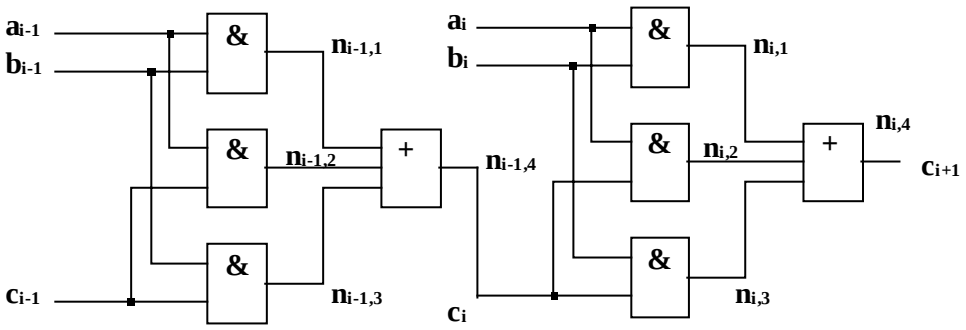


Fig.II.4. A carry chain in the cell C_{i-1} and C_i

Figure II.4 shows a carry chain in the cell C_{i-1} and C_i. First, suppose $k = 2$, and let us consider s/0 at node $\mathbf{n_{i,1}}$. To detect the fault, the path from $\mathbf{n_{i,1}}$ to $\mathbf{n_{i,4}}$ should be sensitized, and 1 should be applied to $\mathbf{n_{i,1}}$ to provoke the fault s/0; then, $\mathbf{c_{i+1}}$ will have an incorrect value, which changes $\mathbf{s_{i+1}}$ in the next cell C_{i+1}. This will corrupt the computation result and result in $k \cdot x \neq x'$. On the other hand, when x' (= $2x$) is applied to the adder and provokes the fault s/0 at $\mathbf{n_{i,1}}$, we can analogously think that x is applied and provokes the fault s/0 at $\mathbf{n_{i-1,1}}$, the node in the previous cell. Therefore,

$$\begin{aligned}\Pr\{k \cdot x \neq x' \mid \text{s/0 at } \mathbf{n_{i,1}}\} = {} & \Pr\{\text{detect s/0 at } \mathbf{n_{i-1,1}}\} + \Pr\{\text{detect s/0 at } \mathbf{n_{i,1}}\} \\ & - \Pr\{\text{detect s/0 at } \mathbf{n_{i-1,1}} \ \&\ \text{detect s/0 at } \mathbf{n_{i,1}}\}\end{aligned} \tag{II.1}$$

However, setting $\mathbf{n_{i-1,1}}$ to 1, provoking the fault at $\mathbf{n_{i,1}}$, and sensitizing $\mathbf{n_{i,1}}$ to $\mathbf{n_{i,4}}$ at the same time is impossible.

$$\Pr\{\text{detect s/0 at } \mathbf{n_{i-1,1}} \ \&\ \text{detect s/0 at } \mathbf{n_{i,1}}\}$$

(II.2)
$$= \Pr\{\text{detect s/0 at } \mathbf{n_{i-1,1}} \ \&\ \text{sensitize path } \mathbf{n_{i-1,1}} \text{ to } \mathbf{n_{i,4}} \ \&\ \text{provoke } \mathbf{n_{i,1}} = 1\}$$
$$= \Pr\{\mathbf{n_{i,4}} = 1 \,\&\, (\mathbf{a_i} = 0 \,\&\, \mathbf{b_i} = 0) \,\&\, (\mathbf{a_i} = 1 \,\&\, \mathbf{b_i} = 1)\} = 0.$$

Also assuming $a_i = b_i = 0.5$,

$$\Pr\{\text{detect s/0 at } \mathbf{n_{i-1,1}}\} = \Pr\{\text{sensitize } \mathbf{n_{i-1,1}} \text{ to } \mathbf{n_{i-1,4}}\} \cdot \Pr\{\mathbf{n_{i-1,1}} = 1\}$$

(II.3)
$$= (1 - a_{i-1}c_{i-1})(1 - b_{i-1}c_{i-1}) \cdot a_{i-1}b_{i-1}$$
$$= \frac{1}{4}(1 - c_{i-1})$$

(II.4) $$\Pr\{\text{detect s/0 at } \mathbf{n_{i,1}}\} = \frac{1}{4}(1 - c_i).$$

Applying the results of (II.2), (II.3) and (II.4) to (II.1), we can obtain

(II.5) $$\Pr\{k \cdot x \neq x' \mid \text{s/0 at } \mathbf{n_{i,1}}\} = \frac{1}{4}(2 - c_{i-1} - c_i).$$

Now, let us consider s/1 at $\mathbf{n_{i,1}}$. We can use the same method as in equ. (II.1).

(II.6)
$$\Pr\{k \cdot x \neq x' \mid \text{s/1 at } \mathbf{n_{i,1}}\} = \Pr\{\text{detect s/1 at } \mathbf{n_{i-1,1}}\} + \Pr\{\text{detect s/1 at } \mathbf{n_{i,1}}\}$$
$$- \Pr\{\text{detect s/1 at } \mathbf{n_{i-1,1}} \ \&\ \text{detect s/1 at } \mathbf{n_{i,1}}\}$$

Unlike the case of s/0, Pr{detect s/1 at $\mathbf{n_{i-1,1}}$ & detect s/1 at $\mathbf{n_{i,1}}$} is not zero.

$$\Pr\{\text{detect s/1 at } \mathbf{n_{i-1,1}} \ \&\ \text{detect s/1 at } \mathbf{n_{i,1}}\}$$
$$= \Pr\{(\mathbf{n_{i-1,1}} = 0 \ \&\ \text{sensitize } \mathbf{n_{i-1,1}} \text{ to } \mathbf{n_{i-1,4}}) \,\&\, (\mathbf{n_{i,1}} = 0 \ \&\ \text{sensitize } \mathbf{n_{i,1}} \text{ to } \mathbf{n_{i,4}})\}$$
$$= \Pr\{(\mathbf{n_{i-1,1}} = 0 \ \&\ \text{sensitize } \mathbf{n_{i-1,1}} \text{ to } \mathbf{n_{i-1,4}}) \,\&\, (\mathbf{a_i} = 0 \text{ or } \mathbf{b_i} = 0)\}$$

(II.7)
$$= \Pr\{\mathbf{n_{i-1,4}} = 0 \,\&\, \mathbf{n_{i,1}} = 0\}$$
$$= \Pr\{\mathbf{n_{i-1,4}} = 0 \,\&\, (\mathbf{a_i} = 0 \text{ or } \mathbf{b_i} = 0)\}$$
$$= (1 - c_i)\frac{3}{4}$$

$$\Pr\{\text{detect s/1 at } \mathbf{n_{i-1,1}}\} = \Pr\{\text{sensitize } \mathbf{n_{i-1,1}} \text{ to } \mathbf{n_{i-1,4}}\} \cdot \Pr\{\mathbf{n_{i-1,1}} = 0\}$$

(II.8)
$$= (1 - \boldsymbol{a}_{i-1}\boldsymbol{c}_{i-1})(1 - \boldsymbol{b}_{i-1}\boldsymbol{c}_{i-1}) \cdot (1 - \boldsymbol{a}_{i-1}\boldsymbol{b}_{i-1})$$
$$= \frac{3}{4} - \frac{1}{2}\boldsymbol{c}_{i-1}$$

(II.9) $$\Pr\{\text{detect s/1 at } \mathbf{n_{i,1}}\} = \frac{3}{4} - \frac{1}{2}\boldsymbol{c}_i$$

Finally, applying the results of (II.7), (II.8) and (II.9) to (II.6), we can obtain,

(II.10) $$\Pr\{\boldsymbol{k} \cdot \boldsymbol{x} \neq \boldsymbol{x}' \mid \text{s/1 at } \mathbf{n_{i,1}}\} = \frac{3}{4} - \frac{1}{2}\boldsymbol{c}_{i-1} + \frac{1}{4}\boldsymbol{c}_i.$$

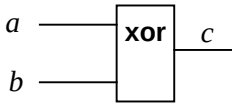
Similarly, we can calculate the probabilities for other nodes.

(II.11) $$\Pr\{k \cdot x \neq x' \mid \text{s/0 at } \mathbf{n}_{i,2}\} = \Pr\{k \cdot x \neq x' \mid \text{s/0 at } \mathbf{n}_{i,3}\} = \frac{1}{4}c_i + \frac{3}{16}c_{i-1}$$

(II.12) $$\Pr\{k \cdot x \neq x' \mid \text{s/0 at } \mathbf{n}_{i,4}\} = \frac{5}{16} + \frac{1}{2}c_i + \frac{1}{8}c_{i-1}$$

(II.13)
$$\Pr\{k \cdot x \neq x' \mid \text{s/1 at } \mathbf{n}_{i,2}\} = \Pr\{k \cdot x \neq x' \mid \text{s/1 at } \mathbf{n}_{i,3}\} = \Pr\{k \cdot x \neq x' \mid \text{s/1 at } \mathbf{n}_{i,3}\}$$
$$= \frac{3}{4} - \frac{1}{2}c_i + \frac{1}{4}c_{i-1}$$

The output of XOR gate keeps the probability of 0.5 for becoming 1 as long as one of the inputs has the probability of 0.5 for 1. In Fig.II.5, one input has probability 0.5 for 1 and the other has p for 1, and as shown in the figure, the probability that output is 1 is still 0.5.



$\Pr\{\mathbf{c}=1\} = \Pr\{\mathbf{a}=1 \& \mathbf{b}=0\} + \Pr\{\mathbf{a}=0 \& \mathbf{b}=1\}$
$= 0.5\,p + 0.5\,(1-p) = 0.5$

Fig.II.5. The output transition probability of XOR gate

We know that signal lines (input and output lines) have $\Pr\{k \cdot x \neq x'\} = 0.75$ from the previous section, and this probability does not change when it propagates through XOR gates; thus, $\Pr\{k \cdot x \neq x'\} = 0.75$ for all the input and output nodes ($\mathbf{a_i}$, $\mathbf{b_i}$, and $\mathbf{s_i}$) of the XOR gate. Table 15 illustrates $\Pr\{k \cdot x \neq x'\}$ in each node in selected cells of a 32-bit adder when the node has stuck fault. The last row averages the probabilities of eight nodes in one particular cell.

Table 15. $\Pr\{k \cdot x \neq x'\}$ in each node in selected cells of the adder when $k = 2$.

	C_0		C_1		C_3		C_7		C_{15}		C_{31}	
	s/0	s/1	s/0	s/1	s/0	s/1	s/0	s/1	s/0	s/1	s/0	s/1
input & $\mathbf{s_i}$	0.750	0.750	0.750	0.750	0.750	0.750	0.750	0.750	0.750	0.750	0.750	0.750
$\mathbf{n_{i,1}}$	0.250	1.000	0.438	0.813	0.297	0.672	0.253	0.628	0.250	0.625	0.250	0.625
$\mathbf{n_{i,2}}$, $\mathbf{n_{i,3}}$	0.000	1.000	0.063	0.813	0.180	0.672	0.216	0.628	0.219	0.625	0.219	0.625
$\mathbf{n_{i,4}}$	0.250	1.000	0.438	0.813	0.578	0.672	0.622	0.628	0.625	0.625	0.625	0.625
total	0.438	0.875	0.500	0.782	0.529	0.711	0.538	0.689	0.539	0.688	0.539	0.688